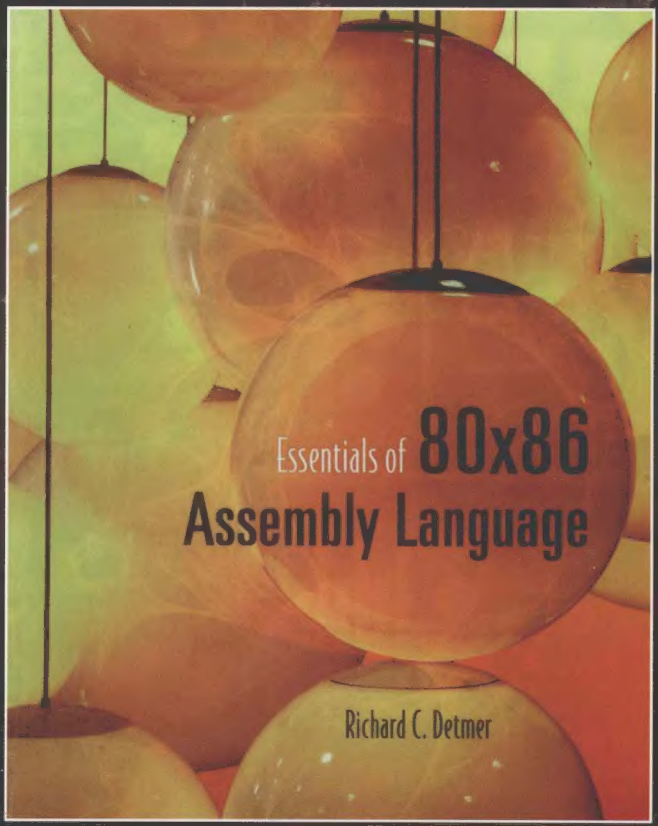
HZ BOOKS
华章教育

Jones and Bartlett

计 算 机 科 学 丛 书

80x86汇编语言基础教程

(美) Richard C. Detmer 著 郑红 陈丽琼 译



Essentials of 80x86 Assembly Language

机械工业出版社
China Machine Press

80x86汇编语言基础教程

学会一门具体的汇编语言对理解计算机体系结构是非常有益的，然而，许多关于计算机组成和体系结构的教材对这方面的知识介绍得不多。本书主要针对Intel 80x86体系结构介绍汇编语言知识，因此既是计算机组成和体系结构课程的很好的补充教材，同时也适合作为单独的汇编语言课程教材。通过本书的学习，学生能够使用微软的MASM汇编器来编译32位的平面存储模式程序，并在微软的Windbg调试器控制下跟踪程序指令的执行，从中了解计算机内部存储器和寄存器内容的变化。本书附带的软件包为编写和调试控制台应用程序提供了很好的环境。

本书特点

- 提供MASM汇编程序的完整软件包、最新的微软链接器、微软的32位全屏调试程序Windbg，并提供一切必要的支持文件。该包为生成和调试控制台应用程序提供了良好环境。
- 提供丰富的图和例子，以及指令“执行前”和“执行后”的情况，帮助学生深入理解本书的内容。
- 内容丰富，包括：数据表示、80x86结构、汇编语言语法、在Windbg中编译和运行程序以及其他的内容。

作者：Patrick Juola
书号：978-7-111-23917-8
定价：42.00元

投稿热线：(010) 88379604
购书热线：(010) 68995259, 68995264
读者信箱：hzjsj@hzbook.com

华章网站 http://www.hzbook.com

网上购书：www.china-pub.com

封面设计：锡杉

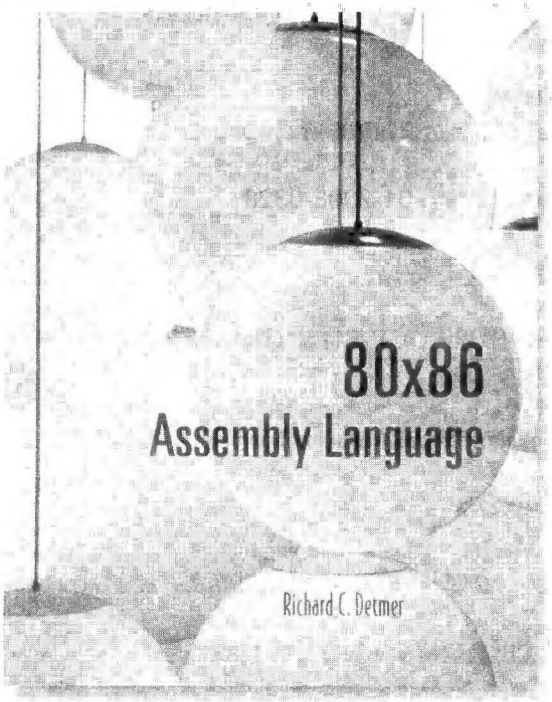
附光盘

上架指导：计算机/汇编语言
ISBN 978-7-111-25382-2
9 787111 253822
定价：35.00元（附光盘）

计　算　机　书

80x86汇编语言基础教程

(美) Richard C. Detmer 著　郑红 陈丽琼 译



Essentials of 80x86 Assembly Language

机械工业出版社
China Machine Press

本书主要针对 Intel 80x86 体系结构介绍汇编语言知识，强调基本的 80x86 整型指令，同时也介绍了浮点型结构，内容基本而完整。通过本书的学习，学生可以使用微软的 MASM 汇编器汇编 32 位平面存储模式程序，并在微软的 Windbg 调试器控制下跟踪程序指令的执行，了解计算机内部存储器和寄存器内容的变化。本书适合用作汇编语言课程的教材，同时也是计算机组成和体系结构课程的很好的补充教材。

Richard C. Detmer：*Essentials of 80x86 Assembly Language* (ISBN 978-0-7637-3621-7).
Copyright © 2007 by Jones and Bartlett Publishers, Inc.
Original English language edition published by Jones and Bartlett Publishers, Inc., 40 Tall Pine Drive, Sudbury, MA 01776.
All rights reserved. No change may be made in the book including, without limitation, the text, solutions, and the title of the book without first obtaining the written consent of Jones and Bartlett Publishers, Inc. All proposals for such changes must be submitted to Jones and Bartlett Publishers, Inc. in English for his written approval.
Chinese simplified language edition published by China Machine Press.
Copyright © 2009 by China Machine Press.

本书中文简体字版由 Jones and Bartlett Publishers, Inc. 授权机械工业出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

版权所有，侵权必究。
本书法律顾问 北京市展达律师事务所

本书版权登记号：图字：01-2008-0328

图书在版编目（CIP）数据

80x86 汇编语言基础教程 /（美）德特默（Detmer，R.C.）著；郑红，陈丽琼译．—北京：机械工业出版社，2009.1
（计算机科学丛书）
书名原文：Essentials of 80x86 Assembly Language

ISBN 978-7-111-25382-2

Ⅰ.8… Ⅱ.①德… ②郑… ③陈… Ⅲ.汇编语言－程序设计－教材 Ⅳ.TP313

中国版本图书馆 CIP 数据核字（2008）第 179283 号

机械工业出版社（北京市西城区百万庄大街 22 号 邮政编码 100037）
责任编辑：刘立卿
北京慧美印刷有限公司印刷
2009年3月第1版第1次印刷
184mm×260mm • 13.75印张
标准书号：ISBN 978-7-111-25382- 2
ISBN 978-7-89482-851-4（光盘）
定价：35.00元（含光盘）

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
本社购书热线：（010）68326294

出版者的话

文艺复兴以降，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域取得了垄断性的优势；也正是这样的传统，使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线，由此而产生的经典科学著作，不仅擘划了研究的范畴，还揭示了学术的源变，既遵循学术规范，又自有学者个性，其价值并不会因年月的流逝而减退。

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇，也是挑战；而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短的现状下，美国等发达国家在其计算机科学发展的几十年间积淀和发展的经典教材仍有许多值得借鉴之处。因此，引进一批国外优秀计算机教材将对我国计算机教育事业的发展起到积极的推动作用，也是与世界接轨、建设真正的世界一流大学的必由之路。

机械工业出版社华章分社较早意识到“出版要为教育服务”。自1998年开始，华章分社就将工作重点放在了遴选、移译国外优秀教材上。经过多年的不懈努力，我们与Pearson，McGraw-Hill，Elsevier，MIT，John Wiley & Sons，Cengage等世界著名出版公司建立了良好的合作关系，从他们现有的数百种教材中甄选出Andrew S. Tanenbaum，Bjarne Stroustrup，Brain W. Kernighan，Dennis Ritchie，Jim Gray，Afred V. Aho，John E. Hopcroft，Jeffrey D. Ullman，Abraham Silberschatz，William Stallings，Donald E. Knuth，John L. Hennessy，Larry L. Peterson等大师名家的一批经典作品，以“计算机科学丛书”为总称出版，供读者学习、研究及珍藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力襄助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作；而原书的作者也相当关注其作品在中国的传播，有的还专程为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近两百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍。其影印版“经典原版书库”作为姊妹篇也被越来越多实施双语教学的学校所采用。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑，这些因素使我们的图书有了质量的保证。随着计算机科学与技术专业学科建设的不断完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都将步入一个新的阶段，我们的目标是尽善尽美，而反馈的意见正是我们达到这一终极目标的重要帮助。华章分社欢迎老师和读者对我们的工作提出建议或给予指正，我们的联系方法如下：

华章网站：www.hzbook.com
电子邮件：hzjsj@hzbook.com
联系电话：（010）88379604
联系地址：北京市西城区百万庄南街1号
邮政编码：100037

HZ BOOKS
华章教育

前 言

许多计算机组成原理或计算机体系结构的书都提供一些通用的知识，但很少或几乎没有涉及亲身实践一个具体的计算机体系结构。本书对于那些希望为学生提供实际操作 Intel 80x86 体系结构经验的老师来说，是一本很好的补充教材。通过本书的学习，学生能够使用微软的 MASM 汇编器（本书附带）汇编 32 位、平面存储模式的程序。本书也可单独作为汇编语言课程的教科书。学生可以在微软的 Windbg 调试器（本书附带）控制下执行程序，通过跟踪程序指令的执行，透视计算机内部来观察存储器和寄存器内容的变化。

本书强调基本的 80x86 整型指令，但是也介绍了浮点型结构。本书将涉及以下主题：

- 80x86 整型数的表示
- 80x86 内存寻址
- 80x86 寄存器
- 汇编语言的语法
- 操作码和指令格式
- 在 Windbg 下汇编和运行程序
- 数据复制指令
- 整型数的加法指令和减法指令
- 整型数的乘法指令
- 整型数的除法指令
- “与”、“或”及“异或”指令
- 移位指令和循环指令
- 无条件转移指令和条件转移指令
- 80x86 堆栈，压入指令和弹出指令
- 子程序包，调用指令和返回指令
- 80x86 浮点数的表示
- 80x86 浮点寄存器
- 部分 80x86 浮点数指令

风格和教学

本书主要通过示例教学。早在第 2 章，本书就给出了一个完整的汇编语言程序，并且在学生能够理解的层次上，仔细地考查了程序的各个部分。随后的章节包含了许多汇编语言代码的例子，同时，对一些新的或者难理解的概念给出了恰当的解释。

本书使用了大量的图表和例子。给出许多“指令执行前”和“指令执行后”的例子来讲解指令。每章节的后面有练习，简答题加深了对所学内容的理解，编程题让学生有机会将书中的内容运用到汇编语言编程中。

软件环境

“标准”80x86 汇编器是微软的宏汇编器（MASM），版本为 6.11。尽管该汇编器生成的代码用于 32 位的平面存储模式编程，非常适合 Windows 95、Windows NT 或者 32 位的微软操作系统环境，但是，与该软件包对应的链接器和调试程序并不适合在这样的系统环境中使用。本书附带一张光盘，包含 MASM（ML）的汇编程序、最新的微软链接器、32 位的全屏调试程序 Windbg（也来自于微软）以及必要的支持文件。该软件包为生成和调试控制台应用程序提供了一个良好的环境。

学习指南

本书的补充内容包括一个教师指南，该指南提供了一些教学提示和许多习题的答案。采用本书作为教材的老师，可向出版中文版的出版社提供申请，索取该教师指南。另外，如果有问题或者建议，可通过 rdetmer@mtsu.edu 联系本书的作者。

目 录

出版者的话

前言

第 1 章 计算机中数的表示 …… 1
1.1 二进制数和十六进制数…… 1
1.2 80x86 存储器 …… 4
1.3 80x86 寄存器 …… 5
1.4 字符编码 …… 8
1.5 有符号整数的二进制补码表示 …… 10
1.6 整数的加减法 …… 13
1.7 本章小结 …… 17
第 2 章 软件工具和汇编语言语法 … 19
2.1 汇编语言语句与文本编辑器 …… 19
2.2 汇编器 …… 23
2.3 链接器 …… 25
2.4 调试器 …… 25
2.5 数据说明 …… 29
2.6 指令操作数 …… 33
2.7 本章小结 …… 35
第 3 章 基本指令 …… 37
3.1 复制数据指令 …… 37
3.2 整数的加法和减法指令 …… 45
3.3 乘法指令 …… 54
3.4 除法指令 …… 62
3.5 本章小结 …… 68
第 4 章 分支与循环 …… 70
4.1 无条件转移指令 …… 70
4.2 条件转移指令、比较指令和 if 结构 … 74
4.3 循环结构的实现 …… 82
4.4 汇编语言的 for 循环 …… 89
4.5 数组 …… 94
4.6 本章小结 …… 99
第 5 章 过程 …… 101
5.1 80x86 堆栈 …… 101
5.2 过程体、调用及返回 …… 107
5.3 参数与局部变量 …… 114
5.4 本章小结 …… 122
第 6 章 位运算 …… 123
6.1 逻辑运算 …… 123
6.2 移位与循环移位指令 …… 131
6.3 本章小结 …… 140
第 7 章 浮点运算 …… 141
7.1 浮点数表示法 …… 141
7.2 80x86 浮点体系 …… 144
7.3 浮点型指令编程 …… 158
7.4 浮点数和嵌入式汇编 …… 171
7.5 本章小结 …… 172
附录 A 十六进制 /ASCII 码转换 …… 174
附录 B 有用的 MS-DOS 命令 …… 175
附录 C MASM 6.11 保留字 …… 176
附录 D 80x86 指令（按助记符排列） …… 180
附录 E 80x86 指令（按操作码排列） …… 197

第 1 章　计算机中数的表示

用 Java 或 C++ 等高级语言编程时，要用到许多不同类型的变量（比如整型、浮点型或者字符型），变量一旦声明，就不需要考虑数据在计算机中是如何表示的。然而，用机器语言编程时，就必须考虑数据存储的位置和方式。本章将讨论 80x86 微处理器数据存储和处理的位置，以及常用的数据表示方式。

1.1　二进制数和十六进制数

计算机用**位**（bit）（二进制数制用 0 或 1 表示不同的电子状态）来表示数值。以 2 为基数，数字 0 和 1 表示**二进制**数。二进制数跟十进制数很相似，但二进制相应的权（从右到左）依次为 1、2、4、8、16 和 2 的更高次幂等，而十进制相应的权为 1、10、100、1000、10000 和 10 的更高次幂等。例如，二进制数 1101 可表示十进制数 13。

1		1		0		1		
one 8	+	one 4	+	no 2	+	one 1	=	13

二进制数很长，在读写时很不方便，比如，八位二进制数 11111010 表示十进制数 250，15 位二进制数 111010100110000 表示十进制数 30 000。而用**十六进制**（基数 16）表示时，大约只需要用到相应二进制表示的四分之一长度的位数。十六进制与二进制的转换很容易，因此，十六进制的表示可以简化二进制的表示。十六进制需要 16 个数字：其中 0、1、2、3、4、5、6、7、8 和 9 与十进制数相同；A、B、C、D、E 和 F 等同于十进制的 10、11、12、13、14 和 15。另外，这几个字母不论是小写还是大写都可用于表示数。

十六进制数中的权值对应 16 的幂，权值从右到左依次是 1、16、256 等等。十六进制数 9D7A 按如下方法计算：

```
  9    ×   4096      36864    [ 4096 = 16^3 ]
+13    ×   256        3328    [ D is 13, 256 = 16^2 ]
+ 7    ×   16          112
+10    ×   1            10    [ A is 10 ]
=40314
```

得出表示的是十进制的 40314。

图 1-1 列举了一些二进制、十六进制和十进制表示的数。读者应该熟记这些数，或者能够很快地推导出来。

上面的例子给出了如何将二进制数或十六进制数转换为十进制数。那么如何将十进制数转换成二进制数或者十六进制数呢？以及如何将二进制数转换为十六进制数呢？下面将介绍如何手工实现转换。通常情况下，可用一个具有二进制、十进制和十六进制转换功能的计算器很容

十进制	十六进制	二进制
0	0	0
1	1	1
2	2	10
3	3	11
4	4	100
5	5	101
6	6	110
7	7	111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

图 1-1 十进制、十六进制和二进制数

易实现转换，这样，数制转换只不过是按一两个键而已。这种计算器可以像十进制那样进行二进制和十六进制的数学运算，而且具有很多其他的用途。注意：这种计算器用七个显示段来显示一个数字。比如，显示小写字母 b 时，看起来像数字 6，其他有些字符也可能难以辨认。

计算器不需要把一个十六进制数转换为对应的二进制形式。事实上，许多二进制数太长，一般的计算器不易显示，因此，每四位二进制数用一个十六进制数表示。其对应的二进制位如图 1-1 的第 3 列所示。如果位数不够四位，必要的时候前面用 0 补充。例如：

$3E8E2_{16}$=11 1011 1000 1110 0010_2

转换数字下标处的 16 和 2 表示基数。如果不易混淆，则这些下标处的数字可以忽略。二进制数补齐位数是为了增强可读性，如十六进制的数字 2 转换为二进制时，最前面用 0 补齐得到 0010。但由于二进制数前面的零不会改变该二进制数的值，所以上例中十六进制数的最高位 3 不需要转换为 0011。

把二进制数转换为十六进制数格式，则与上面的步骤正好相反。把二进制数从右向左每四位进行分隔，每四位二进制数用对应的十六进制数表示，例如：

1011011101001101111 = 101 1011 1010 0110 1111 = 5BA6F

前面介绍了如何将二进制数转换为十进制数。通常，一个很长的二进制数直接转换为十进制数并不采用这种方法。更快的方法是先将二进制数转换为十六进制数，再将该十六进制数转换为十进制数。以上面的 19 位二进制数为例：

1011011101001101111_2
= 101 1011 1010 0110 1111
= $5BA6F_{16}$
= 5×65536+11×4096+10×256+6×16+15×1
= 375407_{10}

下面给出十进制数转换为十六进制数的算法，该算法从右到左依次生成十六进制数位。该

算法用伪代码来描述，本书中其他的所有算法和程序都将采用伪代码描述。

```
until DecimalNumber = 0 loop
    divide DecimalNumber by 16, getting Quotient and Remainder;
    Remainder (in hex) is the next digit (right to left);
    DecimalNumber := Quotient;
end until;
```

示例

以十进制数 5876 转换为十六进制数的过程为例：

- 因为这是一个 until 循环，当第一次执行程序体的时候就进行循环控制条件检查。（为什么不用 while 循环？）
- 16 整除 5876（十进制数）

```
     367     商   新的十进制数的值
16⟌5876
    5872
       4     余数   最右边的十六进制数位
```

当前结果：4

- 367 不等于 0，再用 16 整除。

```
      22     商   新的十进制数的值
16⟌367
     352
      15     余数   生成的第 2 个十六进制数位
```

当前结果：F4

- 22 不等于 0，用 16 整除。

```
       1     商   新的十进制数的值
16⟌22
      16
       6     余数   生成的下一个十六进制数位
```

当前结果 6F4

- 1 不等于 0，用 16 整除。

```
       0     商   新的十进制数的值
16⟌1
       0
       1     余数   生成的下一个十六进制数
```

当前结果 16F4

- 当前的十进制数为 0，循环终止。最后结果为 $16F4_{16}$

练习 1.1

请根据下面给出的数字将表中空白的另外两种进制形式补充完整。

	二进制	十六进制	十进制
1.	100	_____	_____
2.	10101101	_____	_____
3.	1101110101	_____	_____

4.	11111011110	_____	_____
5.	1000000001	_____	_____
6.	_____	8EF	_____
7.	_____	10	_____
8.	_____	A52E	_____
9.	_____	70C	_____
10.	_____	6BD3	_____
11.	_____	_____	100
12.	_____	_____	527
13.	_____	_____	4128
14.	_____	_____	11947
15.	_____	_____	59020

1.2 80x86 存储器

80x86 微机上的**随机存储器**（RAM）逻辑上可以看作是一个“条板单元”的集合，每个单元能存储一个**字节**（8 位）的指令或者数据。每个存储器字节都有一个 32 位的助记符，称为**物理地址**。一个物理地址通常用一个 8 位的十六进制数表示。第一个地址为 00000000_{16}，最后一个地址为无符号数的 $FFFFFFFF_{16}$。图 1-2 给出了一台 PC 中可能的存储器的逻辑图。由 $FFFFFFFF_{16}$ = 4 294 967 295 可知，PC 微机的存储器字节数可达到 4 294 967 296，即 4GB。

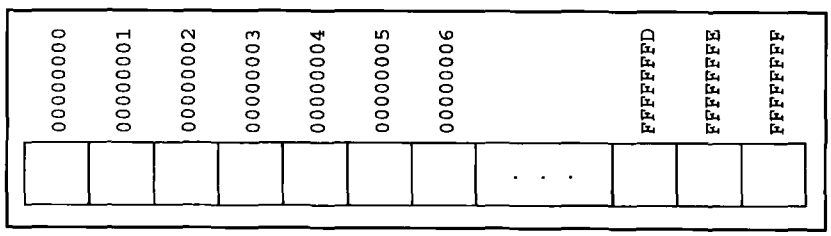


图 1-2 PC 存储器的逻辑图

在 80386 之前，Intel 80x86 处理器系列仅仅能够直接寻址的存储器为 2^{20} 字节，使用 20 位物理地址，通常用 5 个十六进制数表示，范围从 00000 ～ FFFFF。

本书中的汇编语言程序使用**平面存储**模式。这意味着，逻辑上指向存储数据和指令的存储单元的地址实际上是用 32 位的地址编码的。

Intel 80x86 体系结构还提供了**分段存储**模式，早期的 8086/8088 CPU 只用这种模式。在 8086 / 8088 中，PC 存储器可看作是段的集合，每个段是 64KB，以 16 的倍数作为一个段的开始地址。也就是说，如果一个段的起始地址是 00000，另一个段（重叠第一个段）的起始地址就是 16（00010_{16}），下一个段的起始地址是 32（00020_{16}），其他段的起始地址依此类推。一个段的**段号**由其物理地址的前 4 个十六进制数组成。8086/8088 微机中的程序并不能使用 5 位的十六进制数地址，事实上，每个存储单元的定位取决于段号和从该段开始处的一个 16 位的**偏移量**。通常，程序只写出偏移量，段号可通过上下文来推断。偏移量是指从段的第一个字节到要定位地址的距离。在十六进制中，偏移量大小从 0000 到 $FFFF_{16}$。

从 80386 开始，80x86 系列处理器既有 16 位也有 32 位的分段存储模式。段号仍是 16 位长，

但不直接指向存储器中的一个段。事实上，段号仅仅是包含真正 32 位段的起始地址的表中的索引。在 32 位分段模式中，32 位的偏移量加上该段的起始地址可得出内存操作数的实际地址。对编程人员而言，段逻辑上是很有用的：在 Intel 的分段模式下，编程人员通常为代码、数据和系统堆栈分配不同的内存段。80x86 平面存储模式是真正的 32 位分段模式，所有的段寄存器包含相同的值。

事实上，当程序执行时，由程序产生的 32 位地址不一定是某个操作数存储的物理地址，操作系统和 Intel 80x86 CPU 有另外的存储管理层。**分页**（paging）机制用于将程序的 32 位地址映射成物理地址，当程序所产生的逻辑地址超过计算机实际的物理内存空间时，分页机制就非常有用了。如果程序太大而不能全部装入物理内存时，分页机制也可用于将部分程序在需要的时候从磁盘交换到内存中。当用汇编语言编程时，分页机制对用户是透明的。

练习 1.2

1. 假定微机的 RAM 是 256MB，则最后一个字节的 8 位十六进制数的地址是多少？
2. 假定一个微机的视频适配器预定的 RAM 地址是从 000C0000 到 000C7FFF，则其存储空间的字节数是多少？

1.3 80x86 寄存器

早期的 8086/8088 CPU 能够执行 200 多种不同指令。随着 80x86 系列扩展到 80286、80386、80486 以及 Pentium 处理器，CPU 可执行更多的指令。本书探讨如何用这些指令执行程序，从而理解机器级的计算机的性能。其他生产商生产的 CPU 也执行基本相同的指令集，因此，在 80x86 上编写的程序在这些 CPU 上不用改变也可运行。

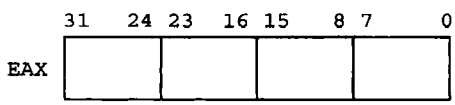
CPU 包含许多**寄存器**。访问每个内部寄存器要比访问 RAM 快得多。应用寄存器主要跟编程人员有关。一个 80x86 CPU（从 80386 开始）有 16 个应用寄存器。常用的指令可在寄存器与存储器间传输数据，也可对存储在寄存器或者存储器中的数据进行操作。所有的寄存器都有名字，一些寄存器有着特定的用途。下面将给出这些寄存器的名字，并详述它们的用途。

寄存器 EAX、EBX、ECX 和 EDX 称为**数据寄存器**或者**通用寄存器**。EAX 有时也是**累加器**，因为它用于存储许多计算的结果。下面是一条使用 EAX 寄存器的指令的例子：

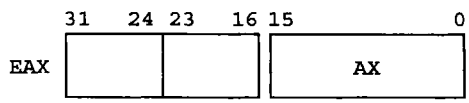
```
add eax , 158
```

将十进制数 158 与 EAX 中已有的数相加，用相加的和取代 EAX 中原来的数。（加法指令和下面提到的其他指令将在第 3 章详细讨论。）

寄存器 EAX、EBX、ECX 和 EDX 都是 32 位长，Intel 转换是以低位的 0 开始，从右向左，对数位转换。因此，如果把每个寄存器看成四个字节，则这些位用数表示为：



寄存器 EAX 整体上按照地址可分成若干部分。低位字位数从 0～15，就是常用的 AX 寄存器。

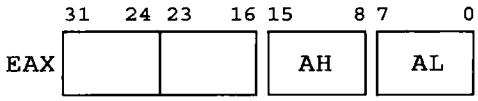


指令

```
sub  ax , 10
```

表示从存储在AX寄存器中的数中减去10，而EAX寄存器的高位数（16～31）没有任何改变。

同样，AX寄存器的低位字节（0～7位）和高位字节（8～15位）分别就是通常所说的AL和AH。



指令

```
mov  ah , 02ah
```

复制2A到8～15位，不改变EAX的其他的任何位。这里的操作数是十六进制而不是十进制。

寄存器EBX、ECX和EDX也有低16位的BX、CX和DX，它们又可按照高位和低位字节分别划分为BH和BL、CH和CL、DH和DL。BH、BL、CH、CL、DH和DL的每位的改变不会改变相应寄存器的其他位。要注意的是EAX、EBX、ECX、EDX的高位字并没有相应的名字——不能只使用名字引用16～31位。

8086到80286处理器有4个16位的通用寄存器，称之为AX、BX、CX和DX。增加“E”后表示“扩展”，将16位扩展成32位的80386寄存器。但是，80386以及后来的体系结构都有效地包含了以前的16位的体系结构。

另外还有4个32位的通用寄存器：ESI、EDI、ESP和EBP。事实上，可以用这些寄存器做算术之类的操作，但通常必须保留它们，用于特定的用途。ESI和EDI寄存器是**索引寄存器**，其中SI代表源索引，DI代表目的索引。它们可用于实现数组索引，并且在某些字符串操作中必须使用，但本书不讨论这些内容。

ESP寄存器是系统栈（内存中保留域）的**栈指针**，它很少直接通过程序来改变，但当数据入栈或者出栈时会改变。ESP寄存器在堆栈的用途之一就是过程（子例程）调用。过程调用指令地址紧跟在存储于栈中的过程调用指令之后，当调用返回时，这条指令地址就可从堆栈中取出。第5章将会详细探讨堆栈以及栈指针寄存器。

EBP寄存器是**基址寄存器**。通常，堆栈中被存取的数据项仅仅是存放在栈顶的数据项。然而，EBP寄存器除了标识栈顶位置外，也经常用于标识栈中的某一个固定位置，因此，在这个固定位置附近的数据可被访问。EBP也用于过程调用，尤其是带有参数的过程调用。

还有6个16位的**段寄存器**：CS、DS、ES、FS、GS和SS。在以前的16位分段存储器模式中，CS寄存器包含有代码段的段号，即当前正在执行的指令所存储的存储器区域。由于一个段是64K长，一个程序的指令集通常在64K的范围内；如果一个程序长度超过64K，则该程序在运行时，需要改变CS的值。同样，DS包含数据段的段号，即数据存储在存储器中的区域。SS寄存器包含有堆栈段的段号，即保留的栈。ES寄存器包含有可用于乘法运算的附加数据段的段号。FS和GS是80386增加的，它们便于访问两个附加数据段。

在平面32位存储器模式中，编程人员不太考虑段寄存器。操作系统给每个CS、DS、ES和SS相同的值。回想一下，这是一个表的入口指针，该表包含段的实际起始地址，也包含程序的大小。因此，当程序随意或者故意对另一个区域进行写操作时，操作系统可能提示错误。但是

这些对编程人员都没有什么关系，编程人员只根据32位地址来考虑。

32位**指令指针**，或称为EIP寄存器，汇编语言的编程人员是不能直接对其进行访问的，CPU必须从存储器中取出要执行的指令，并且，EIP跟踪下一条待取指令的地址。如果是比较老式的、简单的计算机体系结构，下一条待取指令可能也是下一条将要执行的指令。事实上，80x86 CPU在执行前一条指令时，它就取随后要执行的指令了。假设（通常是正确的）：下一次将执行的指令在存储器中是有序紧随（上一条指令）的。如果这种假设证明是错误的，例如，如果执行一个过程调用，则CPU取出存储的指令，设置EIP包含这个过程的偏移量，并且从新的地址取下一条指令。

另外，对于预取指令，80x86 CPU在完成前一条指令的执行前，实际上它就开始执行这个预取指令了。**流水线**（pipelining）使用了这种方法，加快了处理器的有效速度。

最后的寄存器称之为**标志寄存器**（flag register），名为EFLAGS指的就是这种寄存器，但是指令中不使用这个助记符。32位的一些位用于设置80x86处理器的某些特征，其他的位，称为**状态标志位**，表示指令执行的结果，常用的标志寄存器的32位中的有些位的名字在图1-3中给出。标志位的改变取决于所执行的指令。例如，第6位（零标志位ZF）设定为1，因为相加的和为0。其他的标志将在下一节详细讨论。

位	助记符	作用
0	CF	进位标志位
2	PF	奇偶校验标志位
6	ZF	全零标志位
7	SF	符号标志位
11	OF	溢出标志位

图1-3　部分EFLAGS位

总之，80x86 CPU使用16个内部寄存器存储操作数和运算结果，并且跟踪段选择器和段地址，可执行很多指令，图1-4对这些寄存器的使用进行了总结。

名字	长度（位）	使用/说明
EAX	32	累加器，通用；低位字16位AX，可分为AH和AL
EBX	32	通用；低位字16位BX，可分为BH和BL
ECX	32	通用；低位字16位CX，可分为CH和CL
EDX	32	通用；低位字16位DX，可分为DH和DL
ESI	32	源索引；串复制源地址，数组索引
EDI	32	目的索引；目的地址，数组索引
ESP	32	栈指针；栈顶指针
EBP	32	基址指针；栈中引用位置的地址
CS	16	代码段选择器

图1-4　80x86寄存器

名字	长度（位）	使用 / 说明
DS	16	数据段选择器
ES	16	附加段选择器
SS	16	栈段选择器
FS	16	附加段选择器
GS	16	附加段选择器
EIP	32	指令指针；下一条待取指令的地址
EFLAGS	32	标志集；或者状态位

图 1-4 （续）

练习 1.3

1. 画图说明 ECX、CX、CH 和 CL 之间的关系。
2. 标志位 PF 是奇偶校验标志位，计算结果的低位字节的值如果为 1 表示计算结果是偶数，为 0 表示计算结果是奇数。如果计算结果的低字节是 $4E_{16}$，PF 是什么？

1.4 字符编码

字母、数字、标点符号等各种字符在计算机中都可用一个特定的数值来表示。字符编码方式有很多，微机中普遍采用的一种字符编码是美国信息交换标准代码（简称为 ASCII）。

ASCII 用 7 位表示字符，数值从 0000000 到 1111111，它有 128 种组合，可以表示 128 种字符。也可用十六进制数 00 ～ 7F 或者十进制数 0 ～ 127 表示[⊖]。附录 A 给出了 ASCII 的详细列表，在表中可以查到“Computers are fun .”用十六进制表示的 ASCII 码值：

43	6F	6D	70	75	74	65	72	73	20	61	72	65	20	66	75	6E	2E
C	o	m	p	u	t	e	r	s		a	r	e		f	u	n	.

注意：尽管空格字符不可见，但仍然有一个字符编码（十六进制数 20H）。

数字也可以用字符编码表示，例如用 ASCII 表示日期“October 23, 1970”：

4F	63	74	6F	62	65	72	20	32	33	2C	20	31	39	37	30
O	c	t	o	b	e	r		2	3	,		1	9	7	0

其中，日期中的数字字符 23 用 ASCII 码值 32 33 表示，1970 用 31 39 37 30 表示，这与 1.1 节所讲的二进制表示有所不同，上节中 $23_{10}=10111_2$，$1970_{10}=11110110010_2$。计算机使用这两种数字表示方法，其中 ASCII 表示法用于外设输入输出，二进制表示法用于计算机内部计算。

ASCII 的代码看起来似乎是任意指定的，但事实上是遵循某些规范的。大写字母的 ASCII

⊖ 许多计算机使用扩展字符集，它另外增加了十六进制字符 80 ～ FF（十进制 128 ～ 255）。本书不使用扩展字符集。

码值是相邻的，小写字母的 ASCII 码值也是如此。大写字母的编码与其相对应的小写字母的编码仅仅有一位不同，大写字母的第 5 位是 0，而小写字母的第 5 位是 1，其他各位都相同。（通常计算机用位来表示数时，从右到左，最右边的位以第 0 位开始。）例如，

大写字母 M 的编码为 $4D_{16}=1001101_2$

小写字母 m 的编码为 $6D_{16}=1101101_2$

打印输出字符从 20_{16} ～ $7E_{16}$。（空格字符也是打印输出字符。）数字 0，1，…，9 的 ASCII 值分别为 30_{16}，31_{16}，…，39_{16}。

ASCII 码值从 00_{16} 到 $1F_{16}$ 以及 $7F_{16}$ 都是**控制字符**，例如，ASCII 键盘上的 ESC 键的 ASCII 码值是 $1B_{16}$，简称 ESC，表示特殊服务控制，但经常被认为是“escape”的含义。ESC 字符经常与其他字符一起传给外部设备，比如，传给一台打印机，让它执行某种指定的操作。因为这样的字符序列没有标准化，所以本书将不作讨论。

CR 和 LF 是两个非常常见的 ASCII 控制字符，CR（$0D_{16}$）表示返回，LF（$0A_{16}$）表示换行。当按下 ASCII 键盘的 Return 或 Enter 键时，就会产生编码 $0D_{16}$，如果该编码送到 ASCII 显示器，则使光标移到当前行的开始处而不是到新的一行；如果该编码送到 ASCII 打印机，则会使打印头移到当前行的开始处。换行码 $0A_{16}$ 在 ASCII 显示器上会使光标垂直移到下一行或者使打印机将纸向上滚动一行。

使用较少的控制字符有“Form Feed”（$0C_{16}$），该字符使打印机换页；控制字符“Horizontal Tab”（09_{16}）在按下键盘的 Tab 键时生成；“Backspace”（08_{16}）在按下键盘的 Backspace 键时生成；“Delete”（$7F_{16}$）在按下键盘的 Delete 键时生成。注意：Delete 键和 Backspace 键生成的代码不同。响铃（Bell）字符（07_{16}）输出到显示器时会听见响铃声。

许多大型计算机用“扩展二进制编码－十进制信息编码”（简称 EBCDIC）。本书不讨论 EBCDIC 编码。

练习 1.4

1. 每个十六进制数可表示为一个十进制数或两个字符的 ASCII 值，请写出这两种表示。
   a. 2A45
   b. 7352
   c. 2036
   d. 106E
2. 写出下列字符串的 ASCII 值，不要忘记空格和标点符号。返回和换行字符用 CR 和 LF 表示，如果写在一起 CRLF（中间没有空格）表示回车换行功能。
   a. January1 is New Year’s Day. CRLF
   b. George said，“Ouch!”
   c. R2D2 was C3P0’s friend.CRLF[0 是数字 0]
   d. Your name? [问号后输入两个空格]
   e. Enter value: [冒号后输入两个空格]
3. 将下列 ASCII 序列输出到计算机显示器，会显示什么？
   a. 62 6C 6F 6F 64 2C 20 73 77 65 61 74 20 61 6E 64 20 74 65 61 72 73

b. 6E 61 6D 65 0D 0A 61 64 64 72 65 73 73 0D 0A 63 69 74 79 0D 0A

c. 4A 75 6E 65 20 31 31 2C 20 31 39 34 37 0D 0A

d. 24 33 38 39 2E 34 35

e. 49 44 23 3A 20 20 31 32 33 2D 34 35 2D 36 37 38 39

1.5 有符号整数的二进制补码表示

本节详细探讨计算机中数的表示。前面已经介绍了两种表示数的方法，一种是用二进制表示（通常用十六进制表示），另一种是用 ASCII 码表示，但是这两种表示法有两个问题：如何表示一个负数；表示数的有效位是有限的。

假定内存中的数用 ASCII 码存储，一个 ASCII 码通常用一个字节存储，而 ASCII 码长度是 7 位，附加位（左侧或最高位）置 0。为了解决上述表示法中的第一个问题，可在编码中包含一个负数符号。例如：四个字符的 −817 的 ASCII 编码表示就是 2D，38，31 和 37。为解决第二个问题，通常用固定长度的若干字节数，位数不足时，在 ASCII 码的左边用 0 或者空格补充；也可以使用一个长度可变的字节数，但必须规定要表示的数的最后一个数位以 ASCII 码结束，也就是说用一个非数字字符结束。通常，80x86 结构没有什么指令用来处理 ASCII 格式的数字，即使有这样的指令，也很少使用。

计算机内部使用二进制表示数时，80x86 和多数计算机选用固定长度的二进制数运算来解决长度问题，Intel 80x86 系列可用的长度有 8 位（1 字节）、16 位（**1 个字**）[⊖]、32 位（**双字**）和 64 位（**四字**）。

例如，697 的双字长二进制表示为：

$$697_{10} = 1010111001_2 = 00000000000000000000001010111001_2$$

二进制数表示的前面添加 0 是为了使长度是 32 位，该二进制数用十六进制数表示如下：

00	00	02	B9

前面介绍的表示法能够很好地表示非负数和**无符号**数，但不能表示负数。同时，对于任意给定长度都可表示一个最大无符号数，例如一个字节长度，表示的最大无符号数为 FF_{16} 或者 255_{10}。

二进制补码表示法与前面所讲的无符号数表示法很相似，但前者可以表示负数。二进制补码表示数时，应该指定其长度，所以可用“双字长的二进制补码表示法”表示一个数。二进制补码表示非负数与表示无符号数大致相同；也就是说，二进制补码表示数时，前面需要补充很多 0 来确保定长。只有一个限制：对于正数的表示有一个附加位，最左边的一位置 0。如：用字长二进制补码表示最大的正数就是 0111111111111111_2，即 $7FFFF_{16}$ 或者 32767_{10}。

如果根据正数的表示是最左边一位置 0，就推测负数的表示是将最左边一位置 1，其他各位跟对应正数的表示完全相同，那就大错特错了。因为负数的表示要比正数的表示复杂得多，所以不能简单地将最左边的位由 0 改为 1 来表示负数。

一个十六进制的计算器很容易将一个负的十进制数转换为二进制补码表示。例如，计算器

⊖ 其他计算机体系结构一个字的长度可能不是 16 位。

显示 −565，如果按“十六进制”转换键，计算器通常会显示 FFFFFFFDCB（有可能最前面 F 的个数不同）。如果用双字长表示，使用最后的 8 位十六进制数，则为：

FF	FF	FD	CB

或者二进制 1111 1111 1111 1111 1111 1101 1100 1011（注意：最高位用 1 表示负数）。只使用最后四个十六进制数，用字表示，则可表示为：

FD	CB

不用计算器也可实现负数的二进制补码表示。一种方法就是先用十六进制表示这个无符号数，然后用 100000000_{16} 减去这个十六进制数，得到该数的双字长二进制补码表示，十六进制的被减数由 1 后面紧跟表示长度所决定的若干个 0 组成。例如，10000_{16} 就是字长表示补码中的被减数。（字节长度的二进制补码表示的被减数是多少呢？四字长的二进制补码表示的被减数又是多少呢？）以二进制表示，被减数 0 的个数是二进制数位的长度。

示例

用字长的二进制补码表示法表示十进制数 − 76：首先转换无符号十进制数 76 为十六进制数 4C，然后用 10000 减去 4C。

```
1 0 0 0 0
    - 4 C
```

因为 0 不够减 C，所以要从 1000 中借 1，剩下 FFF，即：

```
F  F  F  ¹0
   -  4   C
F  F  B   4
```

借 1 之后做减法就容易多了，相应的数字变为：

$10_{16} - C_{16}=16_{10} - 12_{10} = 4$（十进制或十六进制）

并且 $F_{16} - 4=15_{10} - 4_{10}=11_{10}=B_{16}$

如果已经了解十六进制数的加减法，就没有必要将十六进制数转换为十进制数后再做减法。

1 后面紧跟适当个数的 0 作为被减数，减去某个数的操作称为“**二进制取补**”。这个称谓既包含二进制表示的含义，又包含取补操作的含义，二进制取补操作相当于在十六进制计算器上按下符号转换键。

既然一个给定的二进制补码的表示是固定长度，那么，显然可以存储一个最大范围的数。给定的长度是一个字，则可存储的最大的正数是 7FFF，它是最大的 16 位长的正数，用二进制表示时最高位为 0。十六进制数 7FFF 也就是十进制数 32767。正数用十六进制表示的最高位是 0 到 7（用二进制表示最高位为 0），负数用十六进制表示的最高位为 8 到 F，（用二进制表示最高位是 1）。

怎样把一个二进制补码表示成相应的十进制数呢？首先，确定二进制补码的符号。将正的二进制补码数转换为十进制数，就像任何无符号二进制数转换成十进制数一样，可以手动完

成，也可用有十六进制功能的计算器转换。例如，一个字长的二进制补码数 0D43 表示十进制数 3395。

二进制数位的最高位是 1，即十六进制的 8 到 F 表示的二进制补码数，处理这类负数有些复杂。要注意的是，对一个数求二进制补码，得到结果，再对该结果求其二进制补码，就得到原数。对于长度为双字的数 *N*，通常的代数表达式为：

```
N=100000000 — (100000000 — N)
```

例如：双字长的二进制补码数 FFFFF39E

```
100000000 — (100000000 — FFFFF39E) =100000000 — C62=FFFFF39E
```

这再次说明二进制补码运算与取补运算一致。因此，对二进制位表示的负数求其二进制补码可得到对应的正数（无符号数）。

示例

字长的二进制补码数 E973 表示一个负数，因为符号位（最高位）是 1（E=1110）。取补码可得出相应的正数。

$10000 - E973 = 168D = 5773_{10}$

这说明 E973 表示的十进制数为 −5773。

最高位为 1 的字长二进制补码的范围从 8000 ～ FFFF。转换为十进制数为：

$10000 - 8000 = 8000 = 32768_{10}$

因此，8000 表示的是 −32768。同样地，

$10000 - FFFF = 1$

因此，FFFF 表示的是 −1。由前面得知，字长的二进制补码表示的最大数为十进制正数 32767，因此，其表示的十进制数范围是 −32768 ～ 32767。

用计算器实现将二进制补码表示的负数转换为十进制数是件很棘手的事情。比如，以双字长表示 FFFFFF30，用计算器显示的是 10 位的十六进制数，因此，该数的十位十六进制数的序列为 FFFFFFFF30，前面附加了 6 个 F，然后按计算器的“十进制”转换按钮，计算器上显示的是 −208。

练习 1.5

1. 给出下列每个十进制数的双字长的二进制补码表示：
   a. 3874
   b. 1000000
   c. −100
   d. −55555
2. 给出下列每个十进制数的字长的二进制补码表示：
   a. 845
   b. 15000

c. 100

d. −10

e. −923

3. 给出下列每个十进制数的字节长的二进制补码表示：

a. 23

b. 111

c. −100

d. −55

4. 给出下列每个 32 位双字长的二进制补码数或无符号数所表示的十进制整数：

a. 00 00 F3 E1

b. FF FF FE 03

c. 98 C2 41 7D

d. FF FF FF 78

5. 给出下列每个 16 位字长的二进制补码数或无符号数所表示的十进制整数：

a. 00 A3

b. FF FE

c. 6F 20

d. B6 4A

6. 给出下列每个 8 位字节长的二进制补码数所表示的十进制整数：

a. E1

b. 7C

c. FF

d. 3E

7. a. 给出以字节长的二进制补码形式存储的有符号的十进制整数范围（最小到最大）。

b. 给出以字节形式存储的无符号数的十进制整数的范围。

8. a. 给出以字长的二进制补码形式存储的有符号的十进制整数范围（最小到最大）。

b. 给出以字长形式存储的无符号数的十进制整数的范围。

9. 本节介绍了如何用某个适当的 2 的幂的数作被减数来求一个数的二进制补码。还有一种方法就是以二进制表示该数（选用正确的数位作为表示长度），将每位的 0 变为 1，1 变为 0（也称对 1 取反），最后，对结果加 1（忽略进位）。这两种方法有异曲同工之妙。

1.6 整数的加减法

计算机普遍采用二进制补码表示法，其原因之一在于计算机常用二进制补码存储加减运算的有符号整数，无符号整数的加减法和有符号的加减法类似，这意味着 CPU 不需要增加额外的电路。本小节将讨论整数的加减运算，并介绍可用来确定运算结果是否正确的进位和溢出概念。

首先给出一些加法运算的例子。尽管在本书中 80x86 的指令常用到双字长操作数，为

了简单起见，例子中运用字长操作数。对于字节、字及双字长的操作数，其概念基本相同。80x86 结构系统对有符号数和无符号数使用相同的加法指令。给出的示例中的数都以字长度表示。

首先，0A07 和 01D3 相加。不管采用的是无符号数还是二进制补码表示，这两个数都是正数。右边表示十进制相加的算式。

```
  0A07        2567
+ 01D3      + 467
  0BDA        3034
```

由 $BAD_{16}=3034_{10}$ 可得，该计算结果是正确的。

然后，0206 与 FFB0 相加。显然，正数如同无符号数，但需用二进制补码的有符号数表示，0206 是正数，而 FFB0 为负数，这样就有两种十进制数相加的情况，第一种是有符号数的相加，第二种是无符号数的相加。

```
  0206          518          518
+ FFB0       + (-80)      + 65456
 101B6          438        65974
```

显然，运算结果不唯一。事实上，101B6 是十六进制数表示的 65974，但其不能用长度为 1 个字的无符号数表示（长度为字的无符号数能表示的最大正数为 65535）；如果用有符号数表示，并且忽略最左边的附加位 1，由 101B6 可得到 01B6，01B6 正是十进制数 438 的二进制补码。

现在 FFE7 与 FFF6 相加。如果用有符号数表示，则这两个数是负数。下面也给出有符号十进制数和无符号十进制数相加的表示。

```
  FFE7        (-25)        65511
+ FFF6      +(-10)       + 65526
 1FFDD         -35        131037
```

由于相加的结果值太大，不能用两个字节数表示，但是如果不考虑附加位 1，则 FFDD 就是 −35 的字长的二进制补码表示。

上面的例子中，最后两个加法运算都有一个向高位的**进位**，除去进位后，其他的数字位不是正确的无符号数的结果，但却是正确的二进制补码表示。即使对于有符号数，也不一定总能得到和的正确二进制补码表示。考虑下面两个正数的相加：

```
  483F        18495
+ 645A      + 25690
  AC99        44185
```

这两个加法运算中没有向高位的进位，但有符号数的表示却是错误的，因为 AC99 表示负数 −21351。直观上看，错误的原因在于，求得的和 44185 比用一个字即两个字节长度存储的最大有符号整数 32767 大（见 1.5 节练习 8）。但是，无符号数求和得到的结果却是正确的。

下面的两个有符号数表示的负数相加的例子，也会出现“错误”的结果。

```
  E9FF          (-5633)        59903
+ 8CF0        +(-29456)      + 36080
 176EF           -35089        95983
```

两个数相加有一个进位，但除进位外的剩下四位数字 76EE 不是正确的有符号数的结果，因为 76EE 表示的是正数 30447，并且，-32768 是以字长存储的最小的负数，从直观上判断运算结果出错了。

上面例子出错的原因在于产生了**溢出**。计算机在做二进制加法时，硬件也可判断是否溢出，而且，如果没有溢出，计算机则认为得到的结果是正确的。事实上，计算机做二进制加法时，其运算过程逻辑上从右向左进行各位相加运算，与手动做十进制加法的过程很相似。计算机在逐位相加时，有时会向临近的左边一列产生进位“1”，这个进位会与左边的一列相加的结果一起求和，其他各列相加时依此类推。最特殊的一列是最左边的一列，即符号位，可能有进位到该位，也有可能该符号位进位到“附加位”。符号位进位输出（输出到“附加位”）即前面所谈到的“进位”，以附加的十六进制数 1 表示。图 1-5 给出了出现溢出和没有溢出的情况，从这可以总结出：向“附加位”的进位与向符号位的进位同时有或者同时没有时，不会发生溢出，否则发生溢出。

符号位是否有进位	符号位是否有进位输出	是否溢出
无	无	无
无	有	有
有	无	有
有	有	无

图 1-5　加法运算的溢出情况

下面以二进制形式再次给出上述加法的例子，进位写在两加数的上方。

```
                 111
  0000 1010 0000 0111          0A07
+ 0000 0001 1101 0011        + 01D3
  0000 1011 1101 1010          0BDA
```

该例没有向符号位（第 15 位）的进位，也没有符号位进位输出，所以没有溢出。对第 1、2 和 3 位的进位对溢出没有影响。

```
1 1111 11
  0000 0010 0000 0110          0206
+ 1111 1111 1011 0000        + FFB0
1 0000 0001 1011 0110          101B6
```

该例有向符号位的进位，并且有符号位进位输出，所以没有溢出。

```
1 1111 1111 11    11
  1111 1111 1110 0111          FFE7
+ 1111 1111 1111 0110        + FFF6
1 1111 1111 1101 1101          1FFDD
```

同样，该例既有向符号位的进位也有符号位的进位输出，所以也没有溢出。

```
  1          1111 11
  0100 1000 0011 1111          483F
+ 0110 0100 0101 1010        + 645A
  1010 1100 1001 1001          AC99
```

该例中，有向符号位的进位，但符号位没有进位输出，所以发生了溢出。

```
1    1   11 111
  1110 1001 1111 1111          E9FF
+ 1000 1100 1111 0000        + 8CF0
1 0111 0110 1110 1111         176EF
```

该例子因为没有向符号位的进位，但有符号位的进位输出，所以产生了溢出。

在计算机中，$a-b$ 这样的减法算式，通常是取 b 的二进制补码，然后将其与 a 相加，这就相当于 a 与 $-b$ 相加。例如：十进制减法 $195-618=-423$，

```
  00C3
- 026A
```

可将 −026A 转换为加上 FD96，因为 FD96 是 026A 的二进制补码。

```
  00C3
+ FD96
  FE59
```

十六进制有符号数 FE59 表示十进制数 −432。观察上面的加法算式，有

```
        11      11
  0000 0000 1100 0011
+ 1111 1101 1001 0110
  1111 1110 0101 1001
```

注意，这个加法算式中没有进位，但是做减法时要借位。做减法 $a-b$ 时，如果无符号数 b 比 a 大则要**借位**。计算机硬件通过将减法运算转换为相应的加法运算，根据加法是否产生进位来判断减法是否要借位，如果加法运算没有进位，则对应的减法运算就需要借位，如果加法运算有进位，则对应的减法运算不需要借位（切记“进位”本身意味着“输出”）。

再举一个减法运算的例子：十进制数 985−411 = 574，用字长的二进制补码表示为：

```
  03D9
- 019B
```

可将减 019B 转换为加上 019B 的二进制补码 FE65。即：

```
                 1 1111 1111 1      1
  03D9             0000 0011 1101 1001
+ FE65           +  1111 1110 0110 0101
 1023E             1 0000 0010 0011 1110
```

不考虑附加位 1，十六进制数 023E 表示十进制数 574。这个例子在做加法时有进位，所以相应的减法运算不需要借位。

减法运算也需要定义溢出。如果知道该运算结果已经超出了某种长度（如字长等）表示法所表示的范围，则可以人为断定这个运算结果出错了。而计算机通过对所做的加法运算进行分析来判断相应的减法运算是否产生了溢出。如果加法运算有溢出，则原始的减法运算也会有溢出；如果加法运算没有溢出，则原始的减法运算也不会有溢出。前面所列举的两个减法运算都没有溢出。如果用字长的二进制补码表示 −29123−15447，就会产生溢出。显然，正确的答案− 44570 已经超出了字长的二进制补码数所表示的范围 −32768 ～ 32767，而计算机硬件在做如下运算时

```
  8E3D
- 3C57
```

将减 3C57 计算转换为加上 3C57 的二进制补码 C3A9。

```
                 1    1 11    111   1
  8E3D             1000 1110 0011 1101
+ C3A9           + 1100 0011 1010 1001
 151E6           1 0101 0001 1110 0110
```

由于符号位有进位输出，但没有向符号位的进位，所以产生了溢出。

本小节的例子是以字长的二进制补码来描述数的加减法运算，这种方法也适用于字节的二进制补码、双字的二进制补码或者其他长度的二进制补码的加减运算。

练习 1.6

完成下列字长的二进制补码数的运算。给出每个加减运算操作具体的和与差，并判断是否有溢出。对于加法运算，判断是否有进位，对于减法运算，判断是否有借位。将运算的结果转换成十进制数来核对所做的答案是否正确。

1. 003F + 02A4
2. 1B48 + 39E1
3. 6C34 + 5028
4. 7FFE + 0002
5. FF07 + 06BD
6. 2A44 + D9CC
7. FFE3 + FC70
8. FE00 + FD2D
9. FFF1 + 8005
10. 8AD0 + EC78
11. 9E58 − EBBC
12. EBBC − 9E58
13. EBBC − 791C
14. 791C − EBBC

1.7 本章小结

计算机用电子信号表示所有数据，这些数据可用二进制数（位）来表示，这种表示法可认为是数的二进制表示。数也可写成十进制、十六进制和二进制格式。

80x86 计算机在内存中存储数据和指令。逻辑上内存是一长串地址单元，每个地址单元可

以存储一个字节的信息。寄存器用来操作数据。

大多数计算机用 ASCII 码表示字符，每个字符包括不能打印的控制字符都有一个 ASCII 码值。

整数可用预定长度的二进制补码表示，如一个字节、一个字或双字，一个二进制表示可解释为无符号数或有符号数。

对于无符号数和二进制补码而言，加减运算是一样的，计算机硬件根据运算的情况判断是否有进位或溢出。对于无符号数的运算，通过进位可判断结果是否正确；对于有符号数的二进制补码的运算，通过溢出可判断结果是否正确。

第 2 章　软件工具和汇编语言语法

本章介绍用于创建和执行 80x86 汇编语言程序的软件工具：文本编辑器、汇编器、链接器和调试器，这些工具运行在微软的 Windows 或 MS-DOS 环境。使用文本编辑器可创建汇编语言源代码文件，然后用汇编器将其翻译为目标代码，生成目标代码文件，由链接器执行，最后生成的可执行程序能单独运行。但是本书讨论的程序是在调试器的控制下运行，这样可观察程序执行时每条指令执行的效果。

2.1　汇编语言语句与文本编辑器

汇编语言的源代码文件包含了一系列语句（statement）。大部分语句每行少于 80 个字符，这样的限制便于打印或者在屏幕上显示。但是，MASM 6.1 汇编器允许长达 512 个字符的句子，在每行的行尾用斜杠“\”（除了句末）将一条语句拆分成了多行。

图 2-1 是一个简短但完整的汇编语言程序，这个例子贯穿全章，用来说明基本的汇编语句，以及在调试器控制下如何编辑、汇编、链接和执行程序。

```
; Example assembly language program -- adds 158 to number in memory
; Author:  R. Detmer
; Date:    10/2004

.386
.MODEL FLAT

ExitProcess PROTO NEAR32 stdcall, dwExitCode:DWORD

.STACK  4096            ; reserve 4096-byte stack

.DATA                   ; reserve storage for data
number  DWORD   -105
sum     DWORD   ?

.CODE                           ; start of main program code
_start:
        mov     eax, number     ; first number to EAX
        add     eax, 158        ; add 158
        mov     sum, eax        ; sum to memory

        INVOKE  ExitProcess, 0  ; exit with return code 0

PUBLIC _start                   ; make entry point public

END                             ; end of source code
```

图 2-1　汇编语言程序示例

由于汇编语言本身很难被读懂，所以需要大量的**注释**（comment）。注释可用于任一行语句中，注释以分号“；”开始，直到本行结束都是注释。如果分号是在第一列或者注释是紧跟在一个有效的句子后面，那么整一行都是注释，我们的例子中多数语句都有注释。

汇编语言的语句有3种类型：指令型语句（instruction），指示型语句（directive），宏（macro）。**指令型**指令被翻译成了目标代码（机器代码），在运行时执行。每一条指令都被唯一地翻译成了80x86 CPU可以执行的命令。图2-1所示例子中有以下三条指令：

```
mov eax,number
add eax,158
mov sum,eax
```

第1条指令将内存中符号名为number的双字长的数复制到EAX寄存器中。第2条指令语句是将双字长的数158与到当前EAX寄存器中的双字长的数相加，其结果放到EAX寄存器中。第3条指令语句将EAX寄存器中的双字节数复制到双字节符号名为sum的内存单元。本书用大量篇幅介绍各种80x86指令语句的细节。

一个**指示型**指令告诉汇编器采取某些动作。这种动作并不会产生机器指令，也不会对目标代码有任何影响。例如：在图2-1所示的示例程序中，.386告诉汇编器识别使用32位操作数的80x86指令。.MODEL FLAT告诉汇编器用一个平面存储模式来生成代码。这些指示型指令和一些其他的指示型语句以句点开头，而有些指示型指令不是。

示例程序还有一些指示型指令。指示型指令.STACK 4096告诉汇编器，请求操作系统为系统堆栈保留4096字节的空间。这个系统堆栈用于执行时的过程调用和本地存储。4096字节的堆栈空间对于大多数程序来说是足够大了。

指示型语句.DATA告诉汇编器要定义数据项。指示型语句DWORD要求汇编器为数据在内存中预留双字空间，第1个数据符号名为number，并初始化为FFFFFF97，第2个数据符号名为sum，并给出默认初始值00000000。2.5节将讨论数据定义指示语句的详细内容。

指示型语句.CODE告诉汇编器接下来的语句是可执行语句。PUBLIC告诉汇编器标记“start”程序的入口点在文件外是可见的。最后一条语句是END，告诉汇编器结束汇编。

指示型语句PROTO告诉汇编器过程ExitProcess被定义在示例程序外。ExitProcess是标准的系统函数，它有一个双字参数。语句

```
INVOKE ExitProcess, 0
```

是一个调用过程ExitProcess的指令，并传送0作为参数值。

一个**宏**指令是一系列语句的缩写，它们可以是指令，指示型指令，或者宏指令。汇编器将一个宏指令解释为多个语句，然后再编译这些语句。在示例程序中没有宏语句，尽管INVOKE的行为非常像宏。

一条语句不是简单的注释，语句包含可用来说明语句目的的助记符以及其他三个部分：名字、操作数、注释。这些部分必须以下列顺序排列：

名字 助记符 操作数；注释

例如，一个程序可能包含下面这个语句

```
zeroCount: mov ecx,0; initialize count to zero
```

作为指令型指令，名字部分以“：”结尾。如果作为指示型指令，名字后面不跟冒号。语句中

的助记符用来标识这个语句是特定的指令语句、指示型语句，或者是宏语句。有些语句没有操作数，有些有一个，有些有多个。如果有多个操作数，中间以逗号分隔，也可以加空格。有时候单操作数有多个部分，中间用空格作分隔，看起来好像有多个操作数。

名字域的用处之一是在程序汇编和链接后，指出指令在内存中的地址，其他的指令就可以很容易地找到这个被标识的指令。如果示例中的加法指令需要在程序中循环执行，那么可以这样写：

```
addLoop: add eax,158
```

这条指令就一直执行下去，直到发现一个 jmp（跳转）指令，它是汇编语言版的 goto 语句：

```
jmp  addloop    ; repeat addition
```

注意，jmp 指令的 addLoop 的结尾处没有出现冒号。

高级语言中的循环结构，例如 while 或者 for 在机器语言中不能使用，但是它们可通过使用 jmp 或其他的跳转指令来实现，详细内容将在第 4 章讨论。

有时候只包含一个名字的一行源代码也是很有用的。例如：

```
_start:
```

在示例程序中，标号 _start：将与第一条指令语句 mov 结合，但这更容易被看作是程序的开始。每个程序必须有一个被标识的开始点，为简单起见，通常使用 _start 来标识开始，由于操作系统的特点，_start 的下划线是必须要有的。

在汇编语言中，名字和其他标识性字符是由字母、数字和特殊字符组成的。允许的特殊字符有下划线（_）、问号（？）、美元符号（$）、at 符号（@）。除了 _start 中的下划线，本书很少使用其他特殊符号。名字不能以数字开头，标识符最多可以有 247 个字符，这样容易组成有意义的名字。对于指令型指令助记符、指示型指令助记符、寄存器或者其他对汇编器来说有特殊意义的保留字，微软的 Macro 汇编器不允许用它们作为名字。附录 C 包含了保留标识符的清单。

汇编程序代码通常不易阅读理解。但是，编写的程序终究要被其他人阅读。因此，尽量考虑程序的可读性，形成良好的编码风格，使用小写字母，这些都有助于提高程序的可读性。

回想一下，汇编语言语句包括名字、助记符、操作数和注释部分。一个风格良好的程序从始至终都包含这几个部分。通常将名字放在第一列。助记符从第 12 列开始，操作数从第 18 列开始，注释从第 30 列开始，保持整体的一致性比强调某一列位置更重要。在汇编语言的源文件中可以出现空白行。用空白行来有效地分割汇编语言的不同结构，就好像把文章分割成自然段一样。

汇编语言可以使用小写字母，也可以使用大写字母。通常情况下不区分大小写。可以用标识符来区别大小写，当然，这种情况只在要使用的编程语言对大小写敏感时才适用。混合大小写的代码比全部用大写的或者全部用小写的代码更容易阅读。全部是大写的代码尤其难读。常规做法是大部分代码使用小写字母，只在指示型指令时使用大写字母。本书中的所有程序都遵循这一规则。

大多数命令是在命令提示符（MS-DOS）下输入。可以调用 Windows 的 Start 菜单中的命令

提示符。在打开 MS-DOS 窗口后，使用 CD（改变目录）命令进入到本书复制的软件目录。例如，桌面上 Software 文件夹内的软件，CD 命令触发一个窗口，类似于图 2-2 所示的窗口。通过输入“exit”指令，关闭命令提示符窗口，一些 Windows 版本允许单击标题栏上的按钮“x”来关闭窗口。附录 B 列出了一些常用的 MS-DOS 命令。

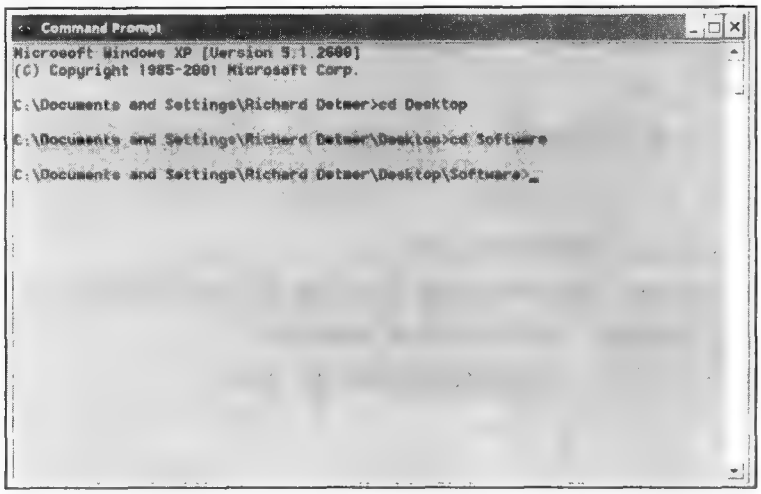


图 2-2 MS-DOS 窗口

文本编辑器、记事本及其他字处理程序（要注意用文本格式保存文件）都可用于写源代码。在命令提示符下可用的 edit 文本编辑器简单易用，可通过输入“edit filename”调用，如果文件存在，就打开该文件，并对文件进行编辑；如果没有，就建立一个新文件。如果要创建汇编语言源文件，那么文件的扩展名必须是 .asm。本章的示例程序存储在文件 example.asm 中，调用命令 `edit example.asm` 后，界面如图 2-3 所示。源代码可根据需要修改，修改完成后，选择菜单栏上的 File 选项下的 Save 选项来保存文件，选择 Exit 选项退出编辑状态。

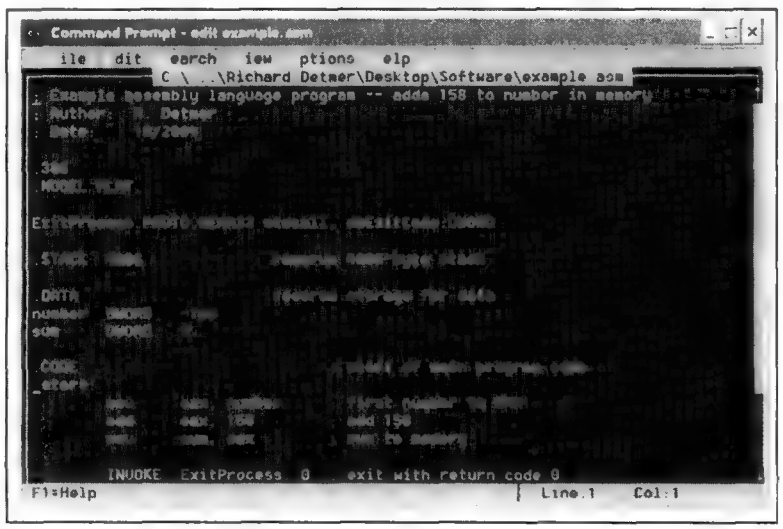


图 2-3 编辑 example.asm

练习 2.1

编辑本章中给出的图 2-1 示例。将代码段中作者的姓名和日期改为当前使用者的名字和使用时的时间。或者复制本书附带的 CD 上的 Software 文件夹，编辑示例代码（将 CD 上的文件拷过来就可以恢复原文件）。

2.2 汇编器

汇编器使用汇编语言源代码，生成 CPU 差不多可执行的目标代码。微软的汇编器 MASM 6.1 称为 ml，ml 程序不仅能汇编程序，它还能链接目标文件。但是，ml 不能链接文件而生成平面存储模式代码，因此，ml 用于汇编，用另外的链接器来链接文件。

回想图 2-1 中的示例程序。假设这个程序存储在文件夹 Software 下的 example.asm 文件中，ml.exe 也存储在文件夹 Software 下，那么用如下命令汇编示例程序：

```
ml  /c  /coff /Fl  /Zi example.asm
```

在命令提示符下输入“/?”，可得到大多数命令的帮助信息。如果输入“ml /?”，汇编器 ml 提示：转换参数 /c 的意思是只需要编译，不需要链接，需要的话，使用一个单独的链接器来链接；转换参数 /coff 说明要生成一个 COFF 格式的目标文件，这时可用链接器生成平面存储模式程序；转换参数 /Fl 说明生成清单文件，它将提供生成的目标代码的信息；最后，转换参数 /Zi 意味着目标代码中要增加调试信息，在调试工具下打开源代码，可以跟踪程序的执行。尽管大多数 MS-DOS 命令大小写都可以，但转换参数是区分大小写的，必须准确地使用。

汇编 example.asm 前，使用命令 dir example.*，列出在 Software 目录下以“example.”开始的文件只有一个：example.asm。汇编 example.exe 后，增加 2 个文件：example.obj 和 example.lst。目标代码文件 example.obj 以后链接。清单文件 example.lst 给出汇编器所做事情的报告。图 2-4 列出了示例程序的部分清单文件。限于页面篇幅，一些注释被删减。使用 edit 或者记事本，可以察看或打印整个文件。

清单文件显示了汇编器是如何建立数据段和翻译指令的。第 1 列表示每条语句的地址，数据起始地址是 00000000。第一个双字的符号名为 number，起始地址也是 00000000，需要 4 个字节。因此，第 2 个双字（符号名为 sum）开始于 00000004。如果有第 3 个数据项，那么它将起始于 00000008。

假定起始地址是 00000000，从指示型语句 .CODE 开始，第 1 条指令语句

```
mov eax, number
```

中，助记符是 mov，操作数是 eax 和 number。助记符 mov 表示复制操作，它将数值从一个地址单元复制到另外一个地址单元。操作数 eax 指示寄存器中的数值表示目的地址，number 代表从数据段 00000000 地址的开始存放的双字。这条指令的操作码是 A1，它表示将从 00000000 地址的开始存放的双字的数值复制到 EAX 寄存器。注意，地址 00000000 后面跟着字母 R。R 表示这是个**重定位**（relocatable）地址，也就是说，在实际运行时，它将被改变到另外的地址。

第 2 条指令

```
add eax,158
```

从地址 00000005 开始，因为第 1 条指令占了 5 个字节。这条指令的助记符是 add，操作数为

```
Microsoft (R) Macro Assembler Version 6.11                    10/10/04 21:48:54
example.asm                                                    Page 1 - 1

                        ; Example assembly language program -- adds 158 to number
                        ; Author:  R. Detmer
                        ; Date:    10/2004

                        .386
                        .MODEL FLAT

                        ExitProcess PROTO NEAR32 stdcall, dwExitCode:DWORD

                        .STACK  4096            ; reserve 4096-byte stack

 00000000               .DATA                   ; reserve storage for data
 00000000 FFFFFF97      number  DWORD   -105
 00000004 00000000      sum     DWORD   ?

 00000000               .CODE                           ; start of main program
 00000000               _start:
 00000000  A1 00000000 R        mov     eax, number     ; first number to EAX
 00000005  05 0000009E          add     eax, 158        ; add 158
 0000000A  A3 00000004 R        mov     sum, eax        ; sum to memory

                                INVOKE  ExitProcess, 0  ; exit with return code 0

 00000016               PUBLIC _start                   ; make entry point public

                        END                             ; end of source code
```

图 2-4 清单文件 example.lst

eax 和 158。汇编器认识到 add 是某个指令的助记符，将执行加法操作。操作数提供汇编器需要的其他信息。

第一个操作数 eax 告诉汇编器用 EAX 寄存器的双字数作为被加数，并且相加之和将存储在 EAX 中。由于第 2 个操作数是一个数字（不同于指定的寄存器或者存储单元中的数），汇编器知道这是一个和 EAX 寄存器中双字数相加的加数。翻译出来的目标代码就是 05 00 00 00 9E，05 表示“将跟在 05 后面的双字的数和存储在 EAX 中的数字相加”。汇编器要将十进制数值 158 转换成双字长度的二进制补码 0000009E。

第 3 条指令语句 mov sum,eax 起始地址 0000000A，因为前 2 条指令语句共占用了 10 个字节。这条指令包含助记符 mov、操作数 sum 和 eax。跟第 1 条指令一样，它将一个 EAX 中双字的数据复制到内存中 sum 标识的地址单元中。这条指令语句操作码是 A3，它表示将 EAX 寄存器中的双字的数复制到内存单元中，地址是从 00000004 后开始，这个地址是重定位的。

如果有第 4 条指令语句，它的起始地址将是 0000000F，因为前 3 条指令语句共占用 15 个字节。从这个示例中，可以得出：每条指令语句被编码为 5 个字节的目标代码。事实上，指令语句可汇编成 1 到 16 个字节的目标代码。指令语句的第 1（或第 2）个字节给出它的操作码，可以简单地在表中查阅相应的操作码，如 mov sum,eax 中的操作码在附录 D 中可以找到，指令中的其余代码有些复杂，本书中将仅讨论部分内容。

练习 2.2

1. 在示例程序中的第 3 条指令后，增加指令 mov ecx,0。汇编修改后的程序，检查清单文件，以确定生成的目标代码是对应新的指令。
2. 虽然 INVOKE 是一个指示型语句，在代码生成上它更像一个宏语句，在图 2-4 的清单文件中没有显示代码，请指出生成了多少字节的代码。（提示：最后一条 mov 指令的起始地址是 0000000A，占用 5 个字节。在 INVOKE 后的第一条语句的址是 00000016。）

2.3 链接器

程序某些部分可以单独汇编。当主程序调用处理各种任务的过程时，分别汇编各个过程可能比较方便。一些过程保存在库中，以便以后使用。链接器把各个单独汇编的模块链接到一个准备载入内存的单个模块中。基本上，它一个接一个地重排单独的目标模块，为组装的载入模块分配地址。当程序实际执行时，这个载入模块被复制到内存中，在载入时，可能要额外地校正地址。

这里用的微软的链接器名为 link。实际系统中可能有不同版本的 link。如果使用本书所带的 CD 复制软件目录，那么就是本书所用的 link。例如，本章下面用到的 link，通过下面的命令提示符调用：

```
link /debug /subsystem:console  /entry:start   /out:example.exe
example.obj kerne132.lib
```

虽然可能会换行，但它是作为一个命令输入的。link 命令将 example.obj 以及来自库文件 kerne132.lib 中所需的过程链接起来，生成输出文件 example.exe。回想一下，example.obj 是通过汇编器 ml 生成的。微软的 kerne132.lib 库包含在本书的软件中，它还有许多过程，包括示例程序所需要的过程 ExitProcess。

在链接命令中有一些转换参数。转换参数 /out 指定了执行程序文件的名称。这个执行程序名必须以 .exe 结束，但不必和正在汇编的程序有相同的基本名。转换参数 /entry:start 标识程序的入口点，它是第 1 条要执行的指令语句的地址。注意：这里并没有使用下划线，虽然 _start 已经是该程序入口点的实际标识。转换参数 /debug 告诉链接器生成调试所必需的文件：example.ilk 和 example.pdb。最后，转换参数 /subsystem:console 告诉链接器生成控制台应用代码，也就是生成一个 DOS 环境下运行的控制台程序。

如果程序没有错误，在 DOS 提示符下就会出现：

```
Microsoft (R)  32-bit incremental linker Version 5.10.7303
Copyright (C)  Microsoft Corp  1992-1997 .All rights reserved.
```

检查目录，会发现文件 example.exe、example.ilk 和 example.pdb 已经建立。

练习 2.3

在计算机上汇编并链接示例程序。

2.4 调试器

调试器允许编程人员控制程序的执行，在每一个指令或者预设的中断点处暂停。当程序

暂停时，编程人员可以检查高级语言形式的变量的内容，以及汇编语言形式的寄存器或者存储器的内容。调试器可用于发现错误，以及“透视”计算机内部，以便发现计算机是如何执行程序的。

本书的软件包有微软的动态调试程序 Windbg，它能用来跟踪汇编语言程序的执行。这是一个有效的查错工具，通过它，也可以了解计算机在机器这一层是如何工作的。

在 DOS 提示符下通过输入“Windbg”，启动调试器，此时会出现一个类似于图 2-5 的窗口。从菜单栏中选择 File，然后打开 Executable，选择 example.exe 文件，或者其他的可执行文件，然后点击 OK，返回到类似图 2-5 的窗口。标题栏中除了增加了 example.exe，Command 窗口中还出现了一些行。为了执行程序，其他的窗口可能需打开。如果这样，只需将 Windbg 窗口返回到前一个窗口。

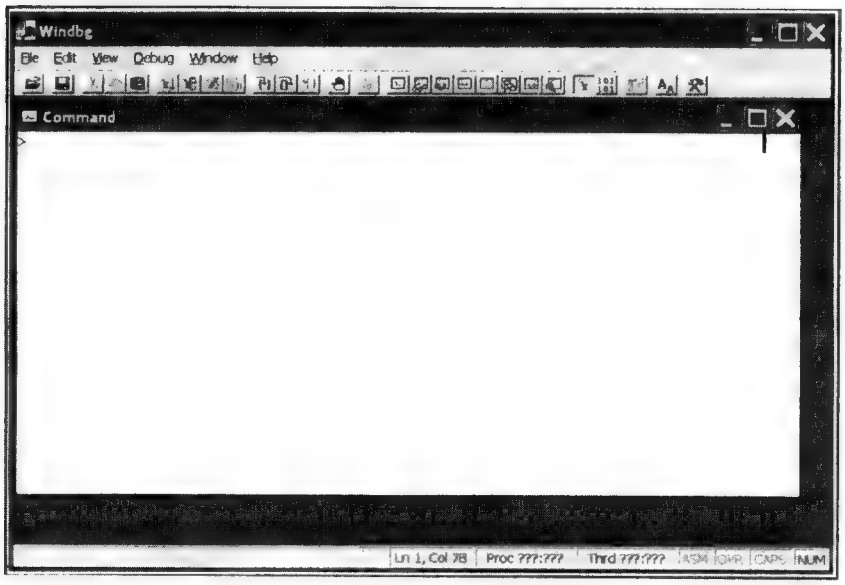


图 2-5 初始的 Windbg 窗口

现在按下 step into 按钮，信息窗口可能出现“没有动态可用来调试的信息”，点击信息窗口中的 OK，然后再点击 step into 按钮。现在源代码就出现在 Windbg 的 Command 窗口后面的子窗口中。将 Command 窗口最小化，选择 View 菜单，然后选择 Registers 子菜单，打开一个窗口，用来显示 80x86 中寄存器的内容。然后选择 View 下 Memory 子菜单，打开一个窗口，用来显示内存的内容，对于该窗口来说，必须输入内存的开始地址。C/C++ 中的取地址符号（&）用来表示地址，数据段的第一项是 number，因此，示例程序使用“&number”就是开始地址。最后调整一下各个窗口的大小，并且重新排列，让屏幕看起来跟图 2-6 显示的那样。

现在，可以用 Windbg“透视计算机内部”。Registers 窗口显示 EAX 的值为 00000000，EBX 的值是 7ffdd000，等等。这些值没有什么意义——都是屏幕左面程序使用后所留下来的。但是，EIP 中的 00401000 给出了执行的第 1 条指令语句 `mov eax,number` 的地址。这条指令以黄色突出显示。ESP 中的值 0012ffc4 也是有意义的，它给出了系统栈顶的地址。

Memory 窗口显示了内存单元开始地址是 00404000。因为要显示的是内存单元 number 的

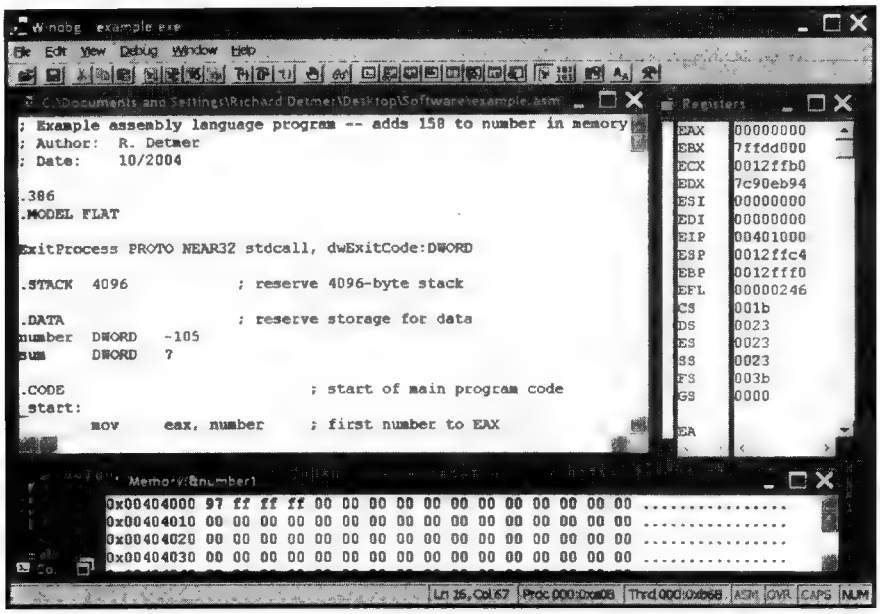


图 2-6 准备运行程序的 Windbg 窗口

地址，因此，现在可以看到 number 运行时的地址了。这个地址的 4 个字节显示为 97 ff ff ff 回想一下，汇编语言清单文件（见图 2-4）显示 number 的值是 FFFFFF97。除了汇编清单显示双字的值以外，这里的值跟看到的是一样的，并且 Windbg 显示的是与实际的值相反的次序存储的字节数。

再次按下 step into 按钮，第 1 条指令语句被执行，并且显示窗口切换，如图 2-7 所示。指令执行改变了寄存器中的值，新的计算结果以红色显示。现在，EAX 寄存器中的值是以红色显

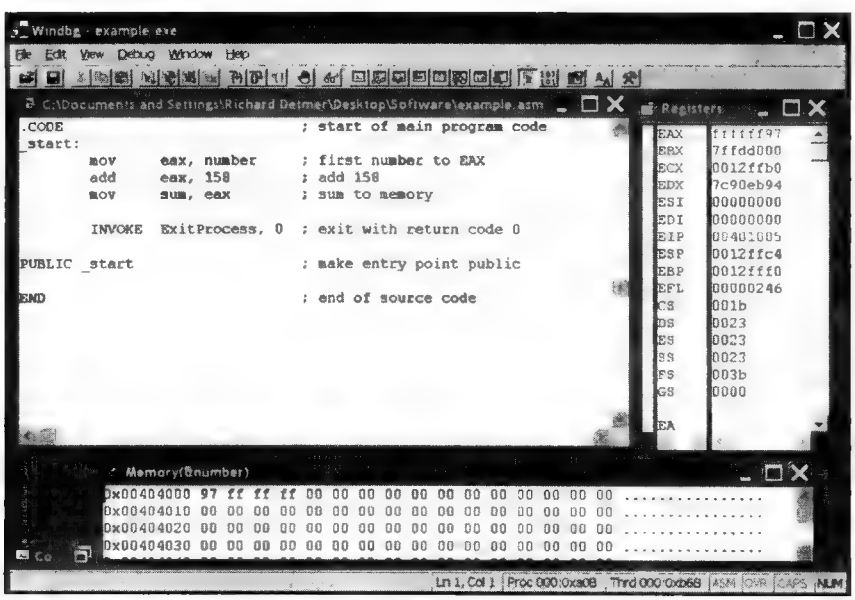


图 2-7 第 1 条指令执行后的 Windbg 窗口

示的 ffffff97，这并不奇怪，因为第 1 个 mov 指令从内存 number 中复制这个值到寄存器 EAX。指令指针 EIP 的值变为 00401005，这是因为第 1 条指令汇编生成 5 个字节的目标代码，所以，先前的地址 00401000 增加 5，指向下一条即将执行的指令。

再次按下 step into 按钮。现在，Windbg 窗口如图 2-8 所示，是第 2 条指令执行后的计算机的状态。EAX 寄存器的值变为 00000035，因为 −105 加上 158，得出 53（十进制）或 35（十六进制）。EIP 寄存器的值是 0040100a，这是第 3 条指令的地址。EFL 寄存器的值也改变了。EFL 上的最后 2 个字节是 0217，或以二进制表示为 0000 0010 0001 0111，其中下划线的二进制位分别是第 11、7、6 和 0 位。回想一下，图 1-3 提到的第 11 位表示溢出位，如果第 11 位是 0，说明没有溢出发生。第 7 位是符号标志位，0 代表用二进制补码数表示的计算结果是正的。第 6 位是 0 标志位，0 表示计算结果不为零。最后，第 0 位是进位标志位，1 表示在进行加法运算时有进位。

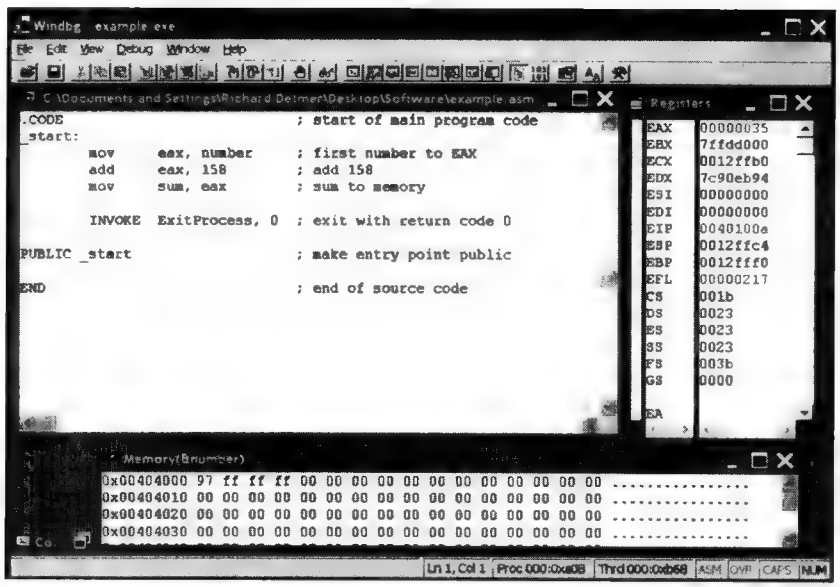


图 2-8 第 2 条指令执行后的 Windbg 窗口

再次按下 step into 按钮，显示出第 3 条指令执行后的计算机状态，如图 2-9 所示。指令指针 EIP 是唯一改变的寄存器，现在，从内存地址 00404004 开始存放数值 35。实际上，这个值是从寄存器 EAX 复制到 00404004 起始的内存单元中的，这个双字的四个字节赋值为 00000035（通过 mov 指令拷贝的值以相反的次序存放），但是因为最后 3 个字节是 0，没有它们对程序也没有影响。如果观察一下 Memory 窗口第 1 行的右边，会看到数字 5 取代了一个句点。如果调试器能把行左边的字节翻译成可打印的 ASCII 字符，那么字符会被显示于行的右边。回想起 35(十六进制)就是字符 5 的 ASCII 代码。当字节没有翻译成可打印的字符时，就使用句点。

再次按下 step into 按钮，调用退出过程 ExitProcess。这次显示窗口没有什么改变，但是 step into 按钮变灰了，因为没有下一步可执行的指令了。此时通常是关闭 Windbg 窗口，虽然也可能选择 Debug（调试）和 Restart（重新开始）重复执行程序。

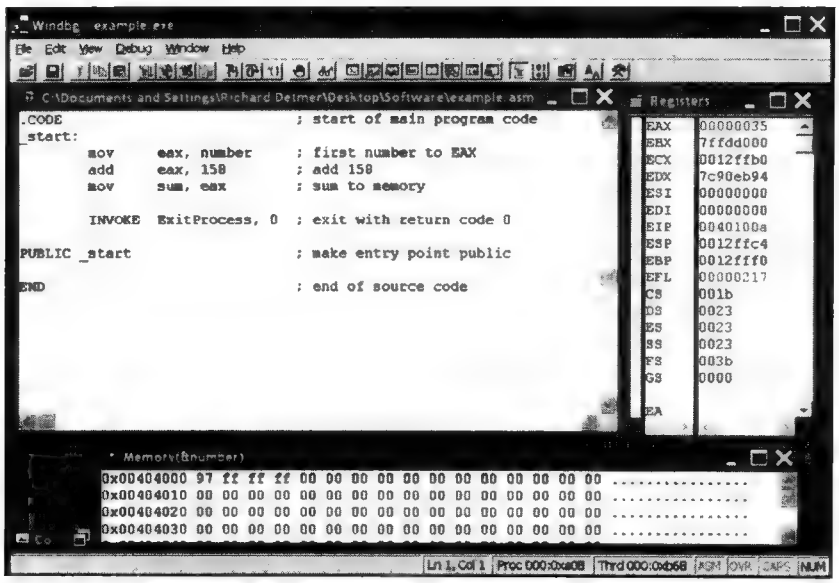


图 2-9　第 3 条指令执行后的 Windbg 窗口

练习 2.4

将示例中的 number 值改为 −253，在第 2 条指令中和 EAX 寄存器中的 74 相加，相加的和存放在 EAX 寄存器中。汇编、链接和执行该程序。说明每条指令执行后，寄存器和内存单元的变化。

2.5　数据说明

这一小节讨论 BYTE、DWORD 和 WORD 等指示型语句中使用的常量操作数的格式，这些讨论多数也适用于指令型语句中的常量，因为简单的常量无论是在指示型语句还是在指令型语句中都是以相同的方式书写的。

数值型的操作数可用十进制、十六进制、二进制或八进制记数法表示。汇编器通常假设数字都是用十进制表示的，除非用其他基准的后缀或者 .RADIX 指示型语句（本书没有使用）改变默认的数字基准。可能用到的后缀如下所示：

后缀	基数	数值系统
H	16	十六进制
B	2	二进制
O 或 Q	8	八进制
无	10	十进制

任何后缀都可用大写或小写的方式编码。如果需要一个八进制常量，字母 Q 比 O 在某些场合阅读起来更清晰。

十六进制数必须以数字开始。例如，为了编码常量值 $A8_{16}$，必须用 0a8h 取代 a8h，否则汇

编器会将 a8h 解释为文件名。

现在举一些例子。指示型语句

```
byte0    BYTE    01111101b
```

在内存单元中预留了 1 字节的空间，并初始化为 7D。这条语句等价于下面的语句：

```
byte0    BYTE    7dh
byte0    BYTE    125
byte0    BYTE    175q
byte0    BYTE    '{'
```

因为 $1111101_2=7D_{16}=125_{10}=175_8$，$7D_{16}$ 是 ASCII 码表示的左大括号“{”。数值系统的选择取决于要使用的常量，当需要把计算结果用做八位序列时，例如第 6 章讨论的逻辑运算，选择二进制来表示数比较合适。如果要将数值当做字符来使用，那么字符用撇号是适当的。

BYTE 指示型语句保留一到多个字节的数据。如果数据计算结果是数值，那么它可能是有符号数，也可能是无符号数。1 个字节存储的无符号的十进数的范围是 0 ～ 255。1 个字节存储的有符号的十进数的范围是 −128 ～ 127。尽管汇编程序允许更大或者更小的数，但是通常会限制 BYTE 语句表示的十进制操作数范围为 −128 ～ 255。下面例子中的注释表示保留字节的初始值：

```
byte1      BYTE    255      ; value is FF
byte2      BYTE    127      ; value is 7F
byte3      BYTE    91       ; value is 5B
byte4      BYTE    0        ; value is 00
byte5      BYTE    -1       ; value is FF
byte6      BYTE    -91      ; value is A5
byte7      BYTE    -128     ; value is 80
```

DWORD 和 WORD 指示型语句的情形类似。DWORD 语句的每个操作数以双字存储。因为 4 字节能存储的有符号数的范围是 −2 147 483 648 ～ 2 147 483 647，或者无符号数的范围是 0 ～ 4 294 967 295，所以限制操作数的范围为 −2 147 483 648 ～ 4 294 967 295。类似地，对于 WORD 语句，每个操作数应该限制在 −32 768 ～ 65 535 范围内。如下例子给出了一些保留双字和字的初始值：

```
double1    DWORD    4294967295   ; value is FFFFFFFF
double2    DWORD    4294966296   ; value is FFFFFC18
double3    DWORD    0            ; value is 00000000
double4    DWORD    -1           ; value is FFFFFFFF
double5    DWORD    -1000        ; value is FFFFFC18
double6    DWORD    -2147483648  ; value is 80000000

word1      WORD     65535        ; value is FFFF
word2      WORD     32767        ; value is 7FFF
word3      WORD     1000         ; value is 03E8
```

```
word4      WORD   0          ; value is 0000
word5      WORD   -1         ; value is FFFF
word6      WORD   -1000      ; value is FC18
word7      WORD   -32768     ; value is 8000
```

上述例子的重点之一，就是不同操作数可用相同的值存储。例如，要注意操作数分别为4294967295和 −1 的两个 DWORD 语句，都生成双字 FFFFFFFF。这个结果可以被认为是无符号数的 4294967295，也可以是有符号数 −1，取决于这个数使用的上下文。

除了数值操作数外，BYTE 语句允许使用单个字符或多个字符组成的字符串作为操作数。撇号（'）或者引号（"）能被用来区别字符或定界字符串。它们必须是成对出现：不能把一个撇号放在左边，而把一个引号放在右边。为了让字符串能够包含某些特殊的字符，撇号定界的字符串可以包含引号，引号定界的字符串可以包含撇号。除非有特别的原因，本书将按照惯例，把单个字符放于一对撇号之间，把字符串放于一对引号之间。

下面的每个 BYTE 语句都是合理的。

```
char1      BYTE   'm'      ; value is 6D
char2      BYTE   6dh      ; value is 6D
string1    BYTE   "Joe"    ; value is 4A 6F 65
string2    BYTE   "Joe's"  ; value is 4A 6F 65 27 73
```

相同的值存储在 char1 和 char2 中。正如前面提醒的，使用的语句必须依赖代码上下文。如果要存储字母 m，那么不需要查阅 ASCII 码 $6D_{16}$——汇编器有内置的 ASCII 表！注意，字符或字符串末尾的定界符、撇号或引号它们本身都不被存储。

BYTE、DWORD 和 WORD 语句可以有多个操作数，它们以逗号分隔。语句

```
dwords     DWORD    10, 20, 30, 40
```

保留 4 个双字空间，存储初始值 000000A、00000014、0000001E 和 00000028。语句

```
string1    BYTE   "Joe"
string1    BYTE   'J', 'o', 'e'
```

存储的值相同。

运算符 DUP 能用来存储未初始化的多个字节或字。它限用于 BYTE、DWORD、WORD 和其他预留存储空间的指示型指令中。语句

```
DblArray   DWORD   100 DUP(999)
```

保留 100 双字的空间，每个都被初始化为 000003E7，这是一个有效的对数组元素进行初始化的方式。如果需要 50 个星号“*”的字符串，那么可用语句：

```
stars      BYTE    50 DUP('*')
```

如果想要 25 个以空格分隔的星号，那么

```
starsAndSpaces BYTE 24 DUP("* "), '*'
```

保留49个字节，并按要求赋初始值。

BYTE、DWORD、WORD及其他语句的操作数可以是算术表达式，也可以是其他运算符的表达式。这些表达式在汇编时由汇编器进行运算，而不是运行时，生成汇编要用的结果。用表达式代替等价值常量是没有什么意义的，但是，有时候这样写有助于代码清晰。如下的语句是等价的，每个都保留一个十六进制的字，其初始值为00000090。

```
gross     DWORD    144
gross     DWORD    12*12
gross     DWORD    10*15 - 7 + 1
```

由BYTE、DWORD或WORD语句定义的每个符号都有一定的长度。汇编器注意到每个符号的长度，并检查以确保这些符号在指令中恰当地使用。例如，如果

```
char      BYTE    'x'
```

用于数据段中，并且

```
mov  EAX, char
```

用于代码段中，寄存器EAX是双字长，而char是单字节长度，那么汇编器将生成错误信息。

微软的汇编器还有一些指示型语句可用来保留存储空间，如存储四字长的QWORD语句，保留10字节整型的TBYTE语句，保留4字节浮点数的REAL4，保留8字节浮点数的REAL8和保留10字节浮点数的REAL10。它也有可用来区别有符号和无符号字节、字和双字的语句，第7章将用到这样的语句。

练习2.5

写出对下面每条语句汇编器生成的初始值。生成的每个字节用两个十六进制数表示。

```
1.  byte1     BYTE    10110111b
2.  byte2     BYTE    33q
3.  byte3     BYTE    0B7h
4.  byte4     BYTE    253
5.  byte5     BYTE    108
6.  byte6     BYTE    -73
7.  byte7     BYTE    'D'
8.  byte8     BYTE    'd'
9.  byte9     BYTE    "John's program"
10. byte10    BYTE    5 DUP("<>")
11. byte11    BYTE    61 + 1
12. byte12    BYTE    'c' - 1
13. dword1    DWORD   1000000
14. dword2    DWORD   1000000b
15. dword3    DWORD   1000000h
```

```
16. dword4    DWORD   1000000q
17. dword5    DWORD   -1000000
18. dword6    DWORD   -2
19. dword7    DWORD   -10
20. dword8    DWORD   23B8C9A5h
21. dword9    DWORD   0, 1, 2, 3
22. dword10   DWORD   5 DUP(0)
23. word1     WORD    1010001001011001b
24. word2     WORD    2274q
25. word3     WORD    2274h
26. word4     WORD    0ffffh
27. word5     WORD    5000
28. word6     WORD    -5000
29. word7     WORD    -5, -4, -3, -2, -1
30. word8     WORD    8 DUP(1)
31. word9     WORD    6 DUP(-999)
32. word10    WORD    100/2
```

2.6 指令操作数

基本的指令操作数有 3 种类型：常量，指定的 CPU 寄存器，引用内存单元。有几种引用内存的方式：本节将讨论较简单的两种方式，其他更复杂的方式将在本书后面章节介绍。

许多指令有 2 个操作数，通常，第 1 个操作数给出操作的目的数，虽然也可能是指源操作数中的一个。第 2 个操作数给出用于运算的源操作数（或者是一个源操作数），而不是目的数。例如，当 `mov al,'/'` 被执行时，字节 2F（斜杠“/”的 ASCII 码表示）被存入 AL 寄存器，以取代之前的字节。第 2 个操作数 `'/'` 说明是常量源操作数。

当 `add eax,number1` 被执行时，寄存器 EAX 得到双字的和，它是 number1 和 eax 中的旧值相加的结果。第 1 个操作数 eax 说明它既是一个双字的源操作数，又是相加结果的目的操作数；第 2 个操作数 number1 指出了其他 2 个双字相加的内存地址。

图 2-10 列出了 Intel 80x86 微处理器的寻址模式，给出了每种模式的数据地址。对于**立即寻址**（immediate mode）模式的操作数，在指令执行前，要使用的计算结果被插入指令中，尽管它曾是个常量[⊖]。通常。计算结果是由汇编器存放，但根据计算结果所处的阶段，它也能通过链接器或载入程序插入，程序员写出包含实际值或符号常量的指令即可。对于**寄存器模式**（register mode）的操作数，要用到的值存在寄存器中。为了表示寄存器模式的操作数，程序员只需简单

方　式	数据所在的地址
立即数	指令操作数就是该数据
寄存器	数据放在某一寄存器中
存储器	数据放在某一内存地址中

图 2-10　80x86 寻址模式

⊖ 人们可以编写**自修改代码**，即代码在执行时可改变它自己的指令。这被认为是非常不好的编程实践。

编码寄存器的名字。寄存器模式的操作数也能指定一个寄存器作为目的单元，但是立即寻址的操作数不能作为目的数。

在下面的例子中，第 1 个操作数是寄存器模式，第 2 个操作数是立即寻址模式。目标代码（取自于汇编清单文件）在这里作为注释列出。对于指令

```
mov al,'/';BO 2F
```

表示斜线“/”的 ASCII 码 2F 是指令的第 2 个操作数，由汇编器放于 AL 寄存器中。对于指令

```
add eax,135; 05 00000087
```

135 的双字长的二进制补码数汇编后存放在指令的最后 4 个字节中。

内存单元地址可通过一些方式计算出来，图 2-11 列出了最常用的两种方式。任何内存模式的操作数都是直接指定内存中使用的数据或指定内存中的目的地址。**直接模式**（direct mode）的操作数有嵌入指令的 32 位地址。通常，程序员编写一个符号，与数据段中 BYTE、DWORD 或 WORD 关联在一起，或者与代码段中的指令关联在一起。与这样一个符号相关的地址将被重定位（relocatable），以便汇编清单可以列出以后可能调整的汇编时的地址。在本章示例程序中，语句

```
mov     sum, eax     ; A3 00000004
```

第 2 个操作数是寄存器模式，第 1 个操作数是直接寻址模式。内存操作数已经被编码在 32 位地址 00000004 单元中，这是数据段中 sum 的偏移量。

存储器寻址方式	数据所在的地址
直接寻址	在内存地址中，其偏移量为指令中的操作数
寄存器间接寻址	寄存器的值作为数据的地址

图 2-11 两种 80x86 内存寻址模式

指令

```
add     eax, [edx]     ; 03 02
```

的第 1 个操作数是寄存器模式，并且第 2 个操作数是**寄存器间接模式**。要注意的是，目标代码不够长，不能存储 32 位内存地址。因此，它使用 EDX 寄存器中的数表示地址，通过这个地址来定位内存中的双字，然后再和 EDX 中的双字相加。换句话说，EDX 中存放的不是第 2 个数，而是第 2 个数的地址。方括号（[]）表示汇编语言中的间接寻址。图 2-12 给出了一些例子，用

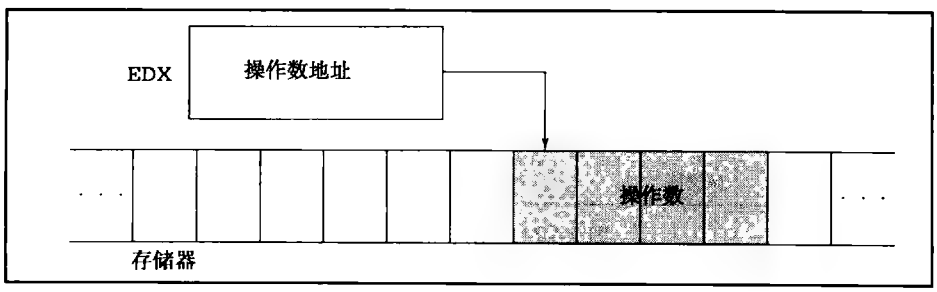


图 2-12 寄存器间接寻址模式

来说明寄存器间接寻址是如何工作的。

任何一个通用寄存器 EAX、EBX、ECX 和 EDX，以及索引寄存器 ESI 和 EDI 都能用于寄存器间接寻址。基准指针 EBP 也可使用，可指向堆栈中的地址，而不是数据段中的地址。尽管在某些特殊场合，栈指针 ESP 可用于寄存器间接寻址，但这样做没有必要。

对于寄存器间接模式，寄存器像高级语言中的指针变量一样使用。寄存器中包含指令用到的数据地址，而不是数据本身。当内存操作数的大小不确定时，运算符 PTR 可用来给出内存单元的大小。例如，对于指令

```
mov [ebx],0
```

汇编器将给出一个错误信息，因为它不能说明目的单元是字节、字还是双字。如果它是字节，那么可以使用

```
mov  BYTE PTR [ebx], 0
```

对于字或双字的目的单元，可分别使用 WORD PTR 或 DWORD PTR。相对地，在指令

```
add     eax, [edx]
```

中，没有必要使用 DWORD PTR [edx]，因为汇编器假定源操作数是双字节的，这是目的操作数 EAX 的大小。

一些指令没有操作数，还有一些指令有单个的操作数。有时候，没有操作数的指令不需要数据去运算，或有一个操作数的指令仅需要一个值。其他时候，一个或多个操作数的地址通过指令表示，且不被编码。例如，乘法的 80x86 指令是 mul，它可能被编码为 mul bh。对于这条指令，只给出了一个操作数：被乘数总是在 AL 寄存器中。（下一章还会进一步介绍这个指令。）

练习 2.6

确定下列指令中每个操作数的模式。假定指令在程序中还包含下列代码：

```
.DATA
value   DWORD       ?
char    BYTE        ?
1.  mov   value, 100
2.  mov   ecx, value
3.  mov   ah, 0ah
4.  mov   eax, [esi]
5.  mov   [ebx], ecx
6.  mov   char, '*'
7.  add   value, 1
8.  add   WORD PTR [ecx], 10
```

2.7 本章小结

本章跟踪了一个汇编语言程序例子：首先用文本编辑器创建它的源代码文件；接着通过汇编器把源代码翻译成目标代码；通过链接器把模块链接在一起组成可执行程序；最后在调试器

Windbg 的控制下执行程序。在这个过程中，可以了解到汇编语言的基本语法和汇编语言程序的结构，也可以学习如何使用 Windbg 调试器。

BYTE、DWORD 或 WORD 语句保留字节、双字节或字的空间，并赋以初始值。

指令操作数有 3 种模式：

- 立即寻址——数据直接列入指令中。
- 寄存器模式——数据放在寄存器中。
- 内存模式——数据在内存中。

内存模式操作数有多种格式，其中有两种是：

直接寻址模式——数据在指令中的地址单元中。

寄存器间接寻址方式——数据在寄存器存放的地址单元中。

第3章 基本指令

本章介绍数据传送指令和整数运算指令，其中数据传送指令主要用于将数据从一个位置复制到另外一个位置。本章尤其强调各种指令所允许的操作数的类型。通过本章的学习，可以了解到如何在存储器和CPU寄存器间复制数据，以及如何在两个寄存器间传送数据。而且，还将了解到如何使用80x86的加、减、乘、除指令，以及这些指令的执行是如何影响标志位的。

3.1 复制数据指令

大多数计算机程序都能将数据从一个位置复制到另外一个位置。对于80x86机器语言，复制工作由mov指令完成，每条mov指令格式如下：

```
mov  destination, source
```

可以从源操作数地址（source）把一个字节、字或双字复制到目的操作数地址（destination），源操作数地址中存储的值不会改变。目的地址必须要求与源地址大小一致。一条mov指令与一条高级语言中的简单赋值语句十分相似。例如，C++或Java中的赋值语句：

```
Count = Number;
```

对应的汇编语言指令如下：

```
mov  Count, ecx          ; Count = Number
```

这里，假设ECX寄存器中的值是Number，而且Count指向存储器中的一个双字。但是，与高级语言的赋值语句相比，mov指令不能执行太多的步骤。例如，赋值语句

```
Count = 3*Number + 1;
```

就不能用一条简单的mov指令来实现。在运算结果的值被复制到目的地址前，需要多条指令来计算等式右边的表达式。

80x86体系结构的局限之一是：并不是所有的源操作数和目的操作数的任意组合都是合理的。特别是，源操作数和目的操作数不能同时在存储器中。指令

```
mov  Count, Number     ; illegal for two memory operands
```

如果Count和Number都指向存储器地址，那么这条指令是非法指令。

所有80x86 mov指令使用同样的助记符来编码。通过查看操作数和助记符，汇编器选择正

确的操作码和其他字节的机器码。

对于一条 mov 指令，源操作数可以是立即数。图 3-1 列出的 mov 指令中有一条指令的源操作数是立即数，目的操作数在寄存器中。图 3-1 中列出了每条指令的字节数和操作码。

目的操作数	源操作数	字节数	操作码
8 位寄存器	字节立即数	2	
AL			B0
CL			B1
DL			B2
BL			B3
AH			B4
CH			B5
DH			B6
BH			B7
16 位寄存器	字立即数	3（包括前缀字节）	
AX			B8
CX			B9
DX			BA
BX			BB
SP			BC
BP			BD
SI			BE
DI			BF
32 位寄存器	双字立即数	5	
EAX			B8
ECX			B9
EDX			BA
EBX			BB
ESP			BC
EBP			BD
ESI			BE
EDI			BF

图 3-1 立即数－寄存器 mov 指令

对于 Intel 80386、80486 和 Pentium 处理器，每条指令的字节数是相同的，这一点同样适用于 8086、8088、80186 和 80286 处理器。但是，8086、8088、80186 和 80286 处理器不支持 32 位寄存器，因此，图 3-1 中的对于 32 位寄存器的指令不适用。

对于 mov 指令，将字或双字立即数复制到寄存器，其操作码都是一样的。80x86 处理器为每一个活动段提供一个**段描述符**（segment descriptor）。该描述符的一位指定操作数是 16 位还是 32 位（默认为 32 位），对于本书中用到的汇编器指示性语句和链接器的选项，该位置为 1 表示 32 位操作数。因此，例如，B8 操作码的意思是复制操作码后的双字立即数到 EAX，而不是

一个单字立即数到 AX。如果编写 16 位指令，那么，汇编器将插入**前缀字节**（prefix byte）66 到目标代码前。通常，前缀字节 66 告诉汇编器，将前缀后的一条指令从默认的操作数长度（32 位或 16 位）转换为可选的长度（32 位或 16 位）。

为了使其更明晰，汇编程序包含以下三条指令：

```
mov     al, 155
mov     ax, 155
mov     eax, 155
```

汇编清单显示的目标代码如下：

```
B0 9B           mov  al, 155
66| B8 009B     mov  ax, 155
B8 0000009B     mov  eax, 155
```

回想一下上述例子，一个立即操作数实际上是汇编为目标代码。每条指令中的立即数 155 都要转换为相应的二进制数，在第一条指令中 155 转换为 9B，在第二条指令中是 009B，在第三条指令中是 0000009B。第一个指令的操作码是 B0，其他两条指令的操作码都是 B8。第二条指令中 66 是前缀字节，告诉汇编程序这条指令操作数长度从 32 位转换为 16 位。

正如第 1 章所讨论的，指令有时会影响 EFLAGS 寄存器中不同的标志位。通常，一条指令可能有如下 3 种影响：

- 不改变标志位；
- 根据指令的执行结果对特定的标志位赋值；
- 某些标志位可能被改变，但是它们的设置无法预测。

所有 mov 指令属于第一种情况。mov 指令不会改变任何标志位。

图 3-2 列出了源操作数为立即数，目的操作数在存储器中的 mov 指令。存储器操作数所占用的字节数由操作数的类型决定。一个直接操作数必须用 32 位地址编码，共 4 个字节。寄存器

目的操作数	源操作数	操作码	字节数
存储器字节	字节立即数	C6	
直接			7
寄存器间接			3
其他			3～8
存储器字	字立即数	C7	
直接			8
寄存器间接			4
其他			4～9
存储器双字	双字立即数	C7	
直接			10
寄存器间接			6
其他			6～11

图 3-2　立即数－存储器 mov 指令

间接操作数由第二个目标代码的三位表示。以后章节将考察其他类型的存储器操作数。16位操作数还需要前缀字节66，这一点没有在表中列出来，因为从技术角度来看，它不是指令的一部分。

假定一个程序包含代码

```
Balance  DWORD  ?
```

在数据段的000000B4位置，并且在代码段中有mov指令

```
mov  Balance, 1000
```

则该汇编程序列表将显示

```
C7 05 000000B4  R      mov Balance, 1000
000003E8
```

分析一下此目标代码，操作数Balance是存储器直接模式，因为它直接指向数据段中的某个位置。图3-2列出目标代码有10个字节，从操作码C7开始，在列表中用两行显示这10个字节，剩余的多数字节也都清楚。存储器中的Balance已经被重定位到32位地址000000B4。直接操作数1000被编码为双字的000003E8。第二个字节被称为mod-reg-r/m字节。根据指令，此字节表示一个寄存器或存储器操作数，对一个或两个寄存器操作数进行编码。对于相同的操作码，这个字节甚至可以区分不同的指令。例如，某些加法和减法指令。这里没有对每条指令进行完整的分析，但将介绍指令的一些使用方法。

mod-reg-r/m字节有三个字段：

- mod（mode），2位；
- reg（register），3位；
- r/m（register/memory），3位。

如果mod = 00，并且r/m = 101，表示直接存储器寻址，在这样指定的mov指令情况下，reg位不使用，并置为000。如果mod-reg-r/m的值为00 000 101，那么在汇编程序清单中出现的就是05。

指令

```
mov    DWORD PTR [ebx], 1000
```

也是一个mov指令的直接存储器寻址的例子，这次是利用寄存器作为间接的目的地址。这条指令在汇编程序清单中显示为：

```
C7 03 000003E8    mov  DWORD PTR [ebx], 1000
```

正如所预期的，这时的操作码是C 7，目标代码是6个字节，立即数是双字的000003E8。这时03编码为00 000 011。同样，reg字段没有被用到。这时r/m字段编码EBX寄存器，一个32位寄存器要么用reg字段编码，要么用r/m字段编码，位模式如图3-3所示。对于r/m字段有一些例外，其中一个就是在最后一个示例中101的意思是直接寻址，而不是寄存器间接寻址中的EBP。同样，ESP也从不会用于寄存器间接寻址，r/m值为100，表示复杂的存储器模式，这将在书中的后面章节讨论。

图3-4列出了其他的80x86 mov指令。这个图引入了一些新的术语。Register 32（32位寄存器）是指32位寄存器EAX、EBX、ECX、EDX、EBP、ESI、EDI或ESP中的一个寄存器；同样，Register 16（16位寄存器）是指16位寄存器AX、BX、CX、DX、SP、BP、SI或DI中的一个

寄存器；同时 Register 8（8 位寄存器）是指 8 位寄存器 AL、AH、BL、BH、CL、CH、DL 或 DH 中的一个寄存器。有些数据传送 mov 指令本书不讨论，比如对一些主要用于系统编程时的寄存器数据传送。

寄存器编码	寄存器
000	EAX
001	ECX
010	EDX
011	EBX
100	ESP
101	EBP
110	ESI
111	EDI

图 3-3　80x86 32 位寄存器编码

目的操作数	源操作数	字节数	操作码
8 位寄存器	8 位寄存器	2	8A
16 位寄存器	16 位寄存器	2	8B
32 位寄存器	32 位寄存器	2	8B
8 位寄存器	存储器字节	2–7	8A
16 位寄存器	存储器字	2–7	8B
32 位寄存器	存储器双字	2–7	8B
AL	直接存储器字节	5	A0
AX	直接字	5	A1
EAX	直接双字	5	A1
存储器字节	8 位寄存器	2 ~ 7	88
存储器字	16 位寄存器	2 ~ 7	89
存储器双字	32 位寄存器	2 ~ 7	89
直接存储器字节	AL	5	A2
直接字	AX	5	A3
直接双字	EAX	5	A3
段寄存器	16 位寄存器	2	8E
16 位寄存器	段寄存器	2	8C
段寄存器	存储器字	2 ~ 7	8E
存储器字	段寄存器	2 ~ 7	8C

图 3-4　其他的 mov 指令

请注意，有时看上去不同的指令却使用相同的操作码。例如，从 8 位寄存器到 8 位寄存器的数据传送和一个存储器字节数到一个 8 位寄存器的数据传送，它们有相同的操作码。在这样的情况下，指令中的第二个字节不仅决定了目标寄存器，而且决定了源寄存器的编码或指出了源存储字节的模式。

假定一个源程序包含语句：

```
dollarSign  BYTE  'S'
```

数据段中的地址是 0000000E，代码段的指令如下：

```
mov  bh, cl            ; register to register move
mov  al, dollarSign    ; memory to register move
mov  bl, dollarSign    ; memory to register move
```

汇编程序为

```
8A F9              mov bh, cl
A0 0000000E R      mov al, dollarSign
8A 1D 0000000E R   mov bl, dollarSign
```

第一条指令很简单。它是一个8位寄存器到8位寄存器的数据传送，在程序中出现的8A是图3-4中显示的操作码的第一项。第二字节又怎么样呢？它编码为11 111 001。第一字段mod=11表示是寄存器到寄存器的传送；reg=111显示目标寄存器为BH，r/m=001显示源操作数为CL。图3-5是图3-3的扩展，它给出了一个完整的寄存器编码列表。

寄存器编码	32位寄存器	16位寄存器	8位寄存器	段寄存器
000	EAX	AX	AL	ES
001	ECX	CX	CL	CS
010	EDX	DX	DL	SS
011	EBX	BX	BL	DS
100	ESP	SP	AH	FS
101	EBP	BP	CH	GS
110	ESI	SI	DH	
111	EDI	DI	BH	

图3-5 80x86寄存器编码

第二条指令是直接存储器到寄存器AL的数据传送。尽管AL是一个8位寄存器，但是没有使用8位寄存器的存储器字节编码。计算机将指令传送给CPU中的累加器进行计算，累加器是一个“特殊”的寄存器，它是唯一的一个可以用于多种运算的寄存器。在80x86中许多寄存器能充当累加器，但是选择累加器可以精简目标代码，5字节的目标代码mov指令简单明了，操作数A0后面是dollarSign的32位地址。

第三条指令有操作码8A和dollarSign的地址的编码。mod-reg-r/m字节被分为00 011 101。正如前面提到的，如果mod=00，并且r/m=101意味着是直接存储器寻址。这次reg位也用到了，它对目的操作数BL进行编码。

对于较老的处理器，访问存储器中数据的指令比访问寄存器中数据的指令执行速度慢。因此，程序员应该尽可能将经常使用的数据保存在寄存器中。

如果第一次看到图3-1、图3-2和图3-4中列举的所有的mov指令，可能认为，能够利用它们将任何源数据复制到目标位置。但是，许多看上去合理的组合不一定可用，其中包括：

- 源操作数和目的操作数都在存储器中的数据传送；
- 操作数长度不一致的数据传送；
- 一次传送多个对象。

有时，可能需要做上述某些操作，下面将讨论如何具体实现。

尽管没有一个mov指令从存储器到存储器间进行数据复制，但是，可以使用两条立即数/寄存器指令来实现这样的工作。例如，对于Count和Number指向的双字长的数据，如下不合法指令

```
mov  Count, Number      ; illegal for two memory operands
```

可以替换为

```
mov eax, Number ; Count : =Number
mov Count, eax
```

每条语句都用了 EAX 累加器和一个直接存储器操作数。除了 EAX 外，也可以使用其他寄存器。但是，使用累加器的每条指令需要 5 个字节，而使用其他相应寄存器的指令需要 6 个字节，从空间效率来考虑，选择 EAX 更好。

如果要在 dblSize 中存储一个双字的数据，在 byteSize 中存储字节型的数据，那么可以用下面的指令：

```
mov  eax, dblSize
mov  byteSize, al
```

还有一种方法，如果在 byteSize 中存储的是一个无符号数或正数，并且将这个数扩展为双字使之与 dblSize 中的值等同，那么可以用下面指令实现：

```
mov  eax, 0
mov  al, byteSize
mov  dblSize, eax
```

请注意，第一条传送指令是确保 EAX 中数据的三位高位字节都是 0，而不是以前运算留下的不确定的值。3.4 节考察其他扩展数的长度的指令。

假设源操作数和目的操作数的声明如下：

```
source    DWORD 4 DUP (?)
dest      DWORD 4 DUP (?)
```

而且希望将全部的四个双字长数据从源地址复制到目标地址，要这样做的一种方法是使用以下 8 条指令：

```
mov    eax, source          ; copy first doubleword
mov    dest, eax
mov    eax, source+4        ; copy first doubleword
mov    dest+4, eax
mov    eax, source+8        ; copy first doubleword
mov    dest+8, eax
mov    eax, source+12       ; copy first doubleword
mov    dest+12, eax
```

source+4 指向的是一个双字地址，即 source 之后的 4 个字节（一个双字长）的地址。由于这 4 个双字长数据在 source 地址起始后的存储器单元中连续存储，因此，source+4 也就指向了第二个双字长数据。当需要复制 40 或 400 个双字长数据时，用这种编码方法，空间效率显然不高。第 4 章将介绍如何设置一个循环来复制多个对象。

80x86 有一个非常有用的 xchg 指令，它可将两个不同地址的数据进行交换，只需要一条简单的指令即可完成，但是高级语言却需要三条指令。假设交换 Value1 和 Value2，在高级语言的设计中，可使用如下语句完成：

```
Temp := Value1 ;        { swap Value1 and Value2 }
Value1 := Value2 ;
Value2 := Temp ;
```

如果 Value1 是存储在 EAX 寄存器中，Value2 是存储在 EBX 寄存器中，ECX 用来暂存 Temp 值，用如下语句可直接实现该算法：

```
mov  ecx, eax        ; swap Value1 and Value2
mov  eax, ebx
mov  ebx, ecx
```

xchg 指令使得代码更简短、明了。

```
xchg  eax, ebx        ; swap Value1 and Value2
```

编写一条指令比三条指令更容易，生成的代码也更容易理解。

图 3-6 列出了 xchg 指令的各种形式。16 位和 32 位指令是一样的，区别仅在于前缀字节，这两种指令都在图中列出了。尽管图中没有列出，但是，如果第二个操作数是寄存器，那么第一个操作数可以是存储器操作数；汇编器能有效地交换操作数的顺序，并可使用图中所列出的形式。

操作数 1	操作数 2	字节数	操作码
8 位寄存器	8 位寄存器	2	86
8 位寄存器	存储器字节	2~7	86
EAX/AX	32/16 位寄存器	1	
	ECX/CX		91
	EDX/DX		92
	EBX/BX		93
	ESP/SP		94
	EBP/BP		95
	ESI/SI		96
	EDI/DI		97
32/16 位寄存器	32/16 位寄存器	2	87
32/16 位寄存器	32/16 位存储器	2~7	87

图 3-6 xchg 指令

在计算机体系结构中，xchg 指令再次表明累加器有时有着特定的作用。这些指令利用一个字节而不是两个字节在累加器和寄存器之间进行数据交换。

注意，不能使用 xchg 指令交换两个存储器操作数。一般情况下，80x86 指令不允许两个存储器操作数。像 mov 指令一样，xchg 指令不更改任何状态标志位；即，xchg 指令执行后，EFLAGS 寄存器的位与指令执行前相同。

练习 3.1

1. 对于如下问题的每部分，在给定的 mov 指令执行时，假设已有执行“前”的值，给出该指令执行“后”的值。

	指令执行之前	执行指令	指令执行之后
a.	EBX: 00 00 FF 75 ECX: 00 00 01 A2	mov ebx, ecx	EBX, CEX
b.	EAX: 00 00 01 A2	mov eax, 100	EAX
c.	EDX: FF 75 4C 2E Value: DWORD –1	mov edx, Value	EDX, Value

d. AX: 01 4B	mov ah, 0	AX
e. AL: 64	mov al, -1	AL
f. EBX: 00 00 3A 4C Value: DWORD ?	mov Value, ebx	EBX, Value
g. ECX: 00 00 00 00	mov ecx, 128	ECX

2. 给出练习题 1 中每条指令的操作码。
3. 编写一个简短的程序，其中包含练习 1 中的每条指令，汇编并检查该程序清单。如果目标代码有一个 mod-reg-r/m 字节，为这个字节的三个字段赋值，根据本节所论述的内容，解释每个字段的值。
4. 对于这个程序的每个部分，假设已有执行“前”的值，给出该指令执行“后”的值。

指令执行之前	执行指令	指令执行之后
a. EBX: 00 00 FF 75 ECX: 00 00 01 A2	xchg ebx, ecx	EBX, ECX
b. EAX: 01 A2 Temp: DWORD -1	xchg Temp, eax	EAX, Temp
c. DX: FF 75	xchg dl, dh	DX
d. AX: 01 4B BX: 5C D9	xchg ah, bl	AX, BX
e. EAX: 12 BC 9A 78 EDX: 56 DE 34 F0	xchg eax, edx	EAX, EDX

5. 给出练习题 4 中每条指令的操作码。
6. 注意，xchg 不能交换存储器中的两个值。编写一系列 mov 或 xchg 指令，交换存储在 Value1 和 Value2 处的双字。假定要使用的任意的 32 位寄存器都是可用的，选择指令操作，以使目标代码字节数最小。

3.2 整数的加法和减法指令

Intel 80x86 微处理器有 add 和 sub 指令，能完成字节、字或双字长操作数的加减运算。操作数可以解释为无符号数或二进制补码的有符号数。80x86 体系结构也提供了 inc 和 dec 指令，用来对单操作数进行加 1 或减 1 操作，以及 neg 指令用来取单操作数的补码。

本节讲到的指令和 3.1 节讲到的 mov、xchg 指令有所不同，add、sub、inc、dec 和 neg 指令都会对 EFLAG 寄存器的标志位进行更新。根据操作数的结果来设置 SF、ZF、OF、AF 标志位的值。例如，如果操作数的结果是负的，那么，符号标志位 SF 将设置为 1；如果操作数的结果为 0，那么，零标志位 ZF 将设置为 1。除了 inc 和 dec 指令，其他指令也会影响进位标志位 CF。

add 指令形式如下：

```
add  destination, source
```

执行加法指令时，源操作数（source）中的整数和目的操作数（destination）中的整数相加，相

加的和将取代目的操作数中原来的值。sub 指令形式如下：

```
sub  destination, source
```

减法指令执行时，目的操作数中的整数减去源操作数中的整数，相减的差将取代目的操作数中原来的值。对于减法，注意，计算得到的差是 destination−source，或者“操作数 1− 操作数 2”。对于加法和减法指令来说，源操作数都是不改变值的。下面举例说明这些指令的执行。

示例

指令执行之前	执行指令	指令执行之后
EAX: 00 00 00 75 ECX: 00 00 01 A2	`add eax, ecx`	EAX [FF][FF][FE][D3] ECX [00][00][01][A2] SF 1 ZF 0 CF 1 OF 0
EAX: 00 00 00 75 ECX: 00 00 01 A2	`sub eax, ecx`	EAX [FF][FF][FE][D3] ECX [00][00][01][A2] SF 1 ZF 0 CF 1 OF 0
AX: 77 AC CX: 4B 35	`add ax, cx`	AX [C2][E1] CX [4B][35] SF 1 ZF 0 CF 0 OF 1
EAX: 00 00 00 75 ECX: 00 00 01 A2	`sub ecx, eax`	EAX [00][00][00][75] ECX [00][00][01][2D] SF 0 ZF 0 CF 0 OF 0
BL: 4B	`add bl, 4`	BL [4F] SF 0 ZF 0 CF 0 OF 0
DX: FF 20 Value中的单字长值	`sub dx, Value` `Value`	DX [00][00] [FF][20] SF 0 ZF 1 CF 0 OF 0
EAX: 00 00 00 09	`add eax, 1`	EAX [00][00][00][0A] SF 0 ZF 0 CF 0 OF 0
Dbl中的双字长值 00 00 01 00	`sub Dbl, 1`	Dbl [00][00][00][FF] SF 0 ZF 0 CF 0 OF 0

加法指令和减法指令设置 SF 标志位与结果的最高位相同，因此，当这些指令用来进行二进制补码整数的加减运算时，如果结果为负，则 SF 标志位为 1。如果结果为 0，那么零标志位 ZF 为 1；如果结果为非零，则零标志位的值为 0。对于进位标志位 CF，它表示加法时的进位以及减法时的借位。溢出标志位 OF 表示溢出，这在第 1 章讨论过。

使用二进制补码数表示有符号数，是因为它不需要为加法或减法指令准备特殊的硬件。同样的电路可以用来对无符号数和二进制补码数进行加法运算。根据操作数的类型不同，标志位会有不同的意义。例如，如果对两个大的无符号整数相加，运算结果的最高位的值为 1，那么，SF 标志位也将置为 1，但它并不表示结果是一个负值，只不过是表示一个相对较大的和而已。对于无符号数相加，如果 CF 标志位的值为 1，则表明结果太大而不能存储在目标位置上；对于有符号数相加，如果 OF 标志位为 1，则表示空间不够而导致溢出。

图 3-7 列出了加法和减法指令。对于每一条加法指令，都有一个相对应的减法指令，它们有相同的操作数类型，相同长度的目标代码，因此，不必将加法指令和减法指令分开列出。对于 80x86，在存储器中只有一个操作数。有些计算机体系结构没有目的操作数是存储器操作数的算术运算指令，但也有一些处理器允许两个操作数都是存储器操作数的算术运算指令。

目的操作数	源操作数	字节数	操作码	
8 位寄存器	8 位立即数	3	80	80
16 位寄存器	8 位立即数	3	83	83
32 位寄存器	8 位立即数	3	83	83
16 位寄存器	16 位立即数	4	81	81
32 位寄存器	32 位立即数	6	81	81
AL	8 位立即数	2	04	2C
AX	16 位立即数	3	05	2D
EAX	32 位立即数	5	05	2D
存储器字节	8 位立即数	3+	80	80
存储器字	8 位立即数	3+	83	83
存储器双字	8 位立即数	3+	83	83
存储器字	16 位立即数	4+	81	81
存储器双字	32 位立即数	6+	81	81
8 位寄存器	8 位寄存器	2	02	2A
16 位寄存器	16 位寄存器	2	03	2B
32 位寄存器	32 位寄存器	2	03	2B
8 位寄存器	存储器字节	2+	02	2A
16 位寄存器	存储器字	2+	03	2B
32 位寄存器	存储器双字	2+	03	2B
存储器字节	8 位寄存器	2+	00	28
存储器字	16 位寄存器	2+	01	29
存储器双字	32 位寄存器	2+	01	29

图 3-7　加法和减法指令

对于 add 和 sub 指令，和 mov 指令一样，累加器还有特殊的指令，当目的操作数是 EAX，AX 或 AL，而源操作数是立即数时，这些指令比其他类似的寄存器指令，需要的目标代码字节数更少。

在图 3-7 中带“+”号的项，一旦知道存储器操作数的类型，就可算出指令的目标代码的总字节数。特别是对直接模式，如果是 32 位地址则需加 4 字节。对于寄存器间接模式，则不需要

其他的目标代码。

请注意，即便目的操作数是一个字或双字，一个立即数的源操作数也可以是一个字节。因为立即操作数通常比较小，这使得目标代码更紧凑。在执行加法或减法指令前，字节长度的操作数被**带符号扩展**（sign-extended）为字或者双字，如果原来的操作数是负数（可看作二进制补码数），那么，用一个或三个 FF 字节扩展成相应的字或双字长的值。一个非负操作数可以简单的用一个或三个 00 字节扩展成相应的字或双字长的值。这两种情况都需复制原来操作数的符号位，使高 8 位或 24 位与符号位相同。

可能有些出乎意料，一些 add 和 sub 指令有同样的操作码。在这种情况下，mod-reg-r/m 字节中的 reg 字段用来区分加法或减法指令。实际上，这些相同的用于加法指令的操作码，在本书后面的章节将涉及到。图 3-8 列出某些操作码的 reg 字段是如何编码的。

	reg 字段								
操作码		000	001	010	011	100	101	110	111
	80, 81 82, 83	ADD	OR	ADC	SBB	AND	SUB	XOR	CMP
	D0, D1 D2, D3	ROL	ROR	RCL	RCR	SHL SAL	SHR		SAR
	F6, F7	TEST		NOT	NEG	MUL	IMUL	DIV	IDIV
	FE, FF	INC	DEC					PUSH (FF only)	

图 3-8 特定操作码的 reg 字段

假设一段程序，在数据段中 Dbl 的地址是 0000001C8，这段程序部分指令为：

```
add  ebx, 1000
sub  ebx, 1000
add  Dbl, 1000
sub  DBl, 1000
add  ebx, 10
```

汇编代码为

```
81 C3 000003E8          add  ebx,1000
81 EB 000003E8          sub  ebx,1000
81 05 000001C8 R        add  Dbl,1000
      000003E8
81 2D 000001C8 R        sub  DBl,1000
      000003E8
83 C3 0A                add  ebx,10
```

请注意，第一条指令和最后一条指令的不同之处。它们都有 32 位目的寄存器操作数 EBX。对这两条指令的操作码，汇编程序都可以用 81，并且立即数 10 编码为 0000000A。但是，对于最后一条指令，汇编程序选择的操作码是 83，这样做的目的是为了精简代码。这条指令执行时，立即数 0A 将扩展为 0000000A。

立即数 1000 不能放在一个字节里，因此，前四条指令将其编码为双字 000003E8。看一下前两条指令中的 mod-reg-r/m 字节：add 指令中的 C3 分解为 11 000 011，sub 指令中的 EB 分

解为 11 101 011。请注意，reg 的值分别为 000 和 101，图 3-8 所列出的 add 和 sub 指令的操作码都是 81，但 reg 字段值不同。回想一下，图 3-5 中 EBX 编码为 011，这是两个指令中 r/m 字段的值，mod 字段中的 11 表示立即操作数。

现在讨论两条源操作数为 Dbl 的指令。其中，直接存储器操作数的地址是 000001C8。加法和减法指令的 mod-reg-r/m 字节分别为 05 和 2D，即分别是 00 000 101 和 00 101 101。区分加法和减法的字段仍然是 reg 字段，mod 为 00，并且 r/m 为 101，这表示直接存储器寻址。

inc 和 dec 指令是有特定用途的加、减指令。通常 1 作为默认的源操作数，它们具有以下形式

```
inc destination
```

和

```
dec destination
```

类似 add 和 sub 这样的指令，inc 和 dec 指令成对出现，而且也要考虑操作数类型、时钟周期、目标代码，图 3-9 对其进行了总结。

目的操作数	字节数	操作码	
		inc	dec
8 位寄存器	2	FE	FE
16 位寄存器	1		
AX		40	48
CX		41	49
DX		42	4A
BX		43	4B
SP		44	4C
BP		45	4D
SI		46	4E
DI		47	4F
32 位寄存器	1		
EAX		40	48
ECX		41	49
EDX		42	4A
EBX		43	4B
ESP		44	4C
EBP		45	4D
ESI		46	4E
EDI		47	4F
存储器字节	2+	FE	FE
存储器字	2+	FF	FF
存储器双字	2+	FF	FF

图 3-9 inc 和 dec 指令

inc 和 dec 指令，把目的操作数看作无符号整数，就像加减指令一样，运算结果将影响 OF、SF 和 ZF 标志位，但是不会改变进位标志位 CF。下面举例说明 inc 和 dec 指令是如何执行的。

示例

指令执行之前	执行指令	指令执行之后
ECX: 00 00 01 A2	`inc ecx`	ECX [00][00][01][A3] SF 0 ZF 0 OF 0
AL: F5	`dec al`	AL [F4] SF 1 ZF 0 OF 0
Count 中的字：00 09	`inc Count`	*Count* [00][0A] SF 0 ZF 0 OF 0
BX: 00 01	`dec bx`	BX [00][00] SF 0 ZF 1 OF 0
EDX: 7F FF FF FF	`inc edx`	EDX [80][00][00][00] SF 1 ZF 0 OF 1

`inc` 和 `dec` 指令在加减计数器做加 1 或减 1 时十分有用，相对于其他加法和减法指令，有时它们需要较少字节的代码。例如，指令

```
add  ecx, 1          ; increment loop counter
```

和

```
inc  ecx             ; increment loop counter
```

在功能上是等效的。`add` 指令需要 3 个字节（因为立即数只需占用 1 个字节，所以占用 3 个字节而不是 6 个字节），而 `inc` 指令只需要 1 个字节。这个例子常用寄存器作为计数器。一般情况下，如果可以预留一个寄存器的话，最好把计数器的值存放在寄存器中。

`neg` 取补指令或者取二进制数补码数，它只有一个操作数。如果正数取补，其结果为负数；对负数取补其结果为正数，而零仍然还是零；`neg` 指令都有如下形式

```
neg  destination
```

图 3-10 给出了 `neg` 所允许的操作数类型。

目的操作数	字节数	操作码
8 位寄存器	2	F6
16 位寄存器	2	F7
32 位寄存器	2	F7
存储器字节	2+	F6
存储器字	2+	F7
存储器双字	2+	F7

图 3-10 `neg` 指令

这里有四个例子来说明 neg 指令是怎样执行的。每种情况下，取补“后”的值正好是取补“前”的二进制补码数。

示例

指令执行之前	执行指令	指令执行之后
EBX: 00 00 01 A2	neg ebx	EBX [FF][FF][FE][5E] SF 1 ZF 0
DH: F5	neg dh	DH [0B] SF 0 ZF 0
Flag 中的字：00 01	neg Flag	*Flag* [FF][FF] SF 1 ZF 0
EAX: 00 00 00 00	neg eax	EAX [00][00][00][00] SF 0 ZF 1

下面是一个完整的示例，这里使用了一些新的指令。这个程序要求输入 x、y、z 三个值，这三个未知数以双字类型放在存储器中，计算表达式 −（x+y−2z+1）的值，并将结果存入 EAX。设计实现如下：

```
result := x;
add y to result, giving x + y;
temp := z;
temp := temp + temp, to get 2*Z
subtract temp from result, giving x + y – 2*z;
add 1 to result, giving x + y – 2*z + 1;
negate result, giving - (x + y – 2*z + 1);
```

写汇编语言程序时，需要合理安排使用寄存器和存储器。在该程序中，要求将计算结果存入 EAX 中，因此唯一要决定的是 temp。这是一个不错的选择，因为和 EAX 相近的寄存器 EBX 没有分配使用。

图 3-11 列出了程序源代码，本程序按照程序意图设计，程序中的注释也同样来自于设计。如果程序中的注释只是简单地重复指令助记符，这对理解程序没有多大意义。注意，2*z 的值通过 z 加 z 得出，下一节将介绍乘法，但是计算 2*z 使用加法计算效率更高。

通过简单的数据预测一下结果，进而测试一下程序。在该程序中只有一组数据，用计算器计算表达式 −（35+47−2*26+1）得到结果为 −31，或者用一个二进制的双字的补码表示为 FFFFFFE1。如果逐句运行程序走查到末尾，Windbg 显示如图 3-12 所示。正确结果存储在 EAX 中。

Windbg 内存显示如图 3-12 所示，存储单元从 x 开始，因此从 1 到 12 个字节是双字，它包含 35、47 和 26。Windbg 允许用户在任何断点更改存储器或寄存器中的值。除了执行第一条指令前 x 的值改为了 −2，图 3-13 与图 3-12 其他都一样。这次表达式 −（−2+47−2*26+1）的结

```
; program to evaluate the expression - (x + y - 2z + 1)
; for doubleword values stored in memory, leaving the result in EAX
; author:  R. Detmer
; date:  revised 5/2005

.386
.MODEL FLAT

ExitProcess PROTO NEAR32 stdcall, dwExitCode:DWORD
.STACK  4096                ; reserve 4096-byte stack

.DATA                       ; reserve storage for data
x           DWORD  35
y           DWORD  47
z           DWORD  26

.CODE                               ; start of main program code
_start:
            mov    eax, x           ; result := x
            add    eax, y           ; result := x + y
            mov    ebx, z           ; temp := z
            add    ebx, ebx         ; temp := 2*z
            sub    eax, ebx         ; result := x + y - 2z
            inc    eax              ; result := x + y - 2*z + 1
            neg    eax              ; result := - (x + y - 2*z + 1)

            INVOKE ExitProcess, 0   ; exit with return code 0

PUBLIC _start                       ; make entry point public
END                                 ; end of source code
```

图 3-11　计算 −(x+y−2z+1) 的程序

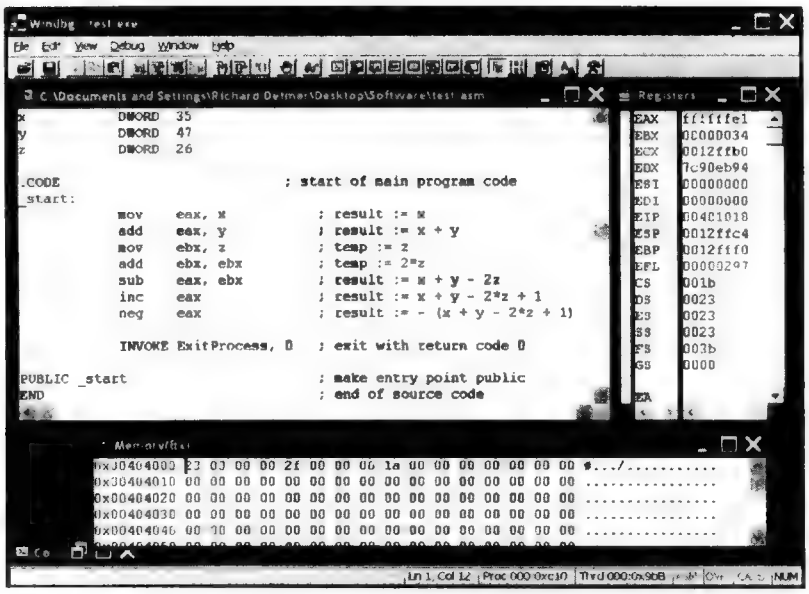


图 3-12　示例程序的执行

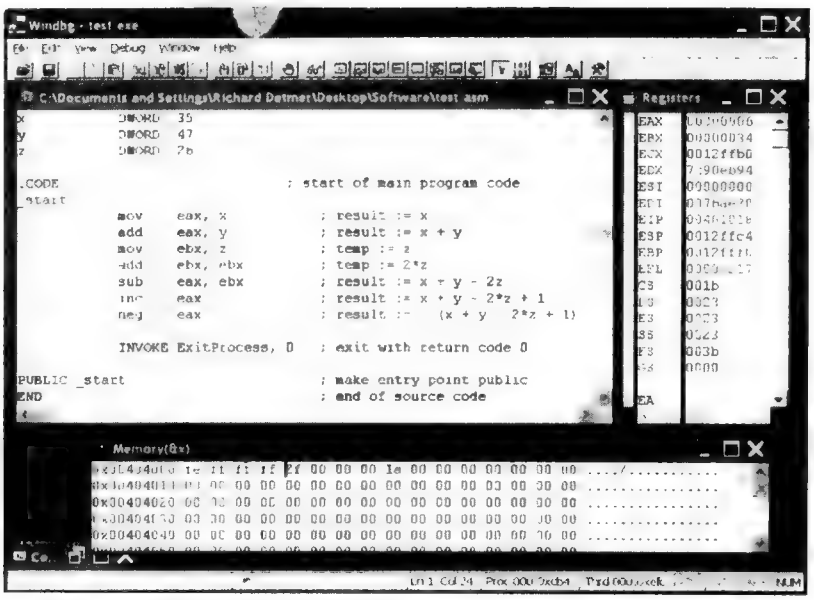


图 3-13 改变存储器值后示例程序的执行

果应该为 6，结果存储在 EAX 中。本书没有讨论输入 / 输出，它们是操作系统的基本功能，但通过修改存储器或寄存器的内容，可以模拟输入。在练习中，这种情况很少出现。

对于大于双字的整型数据的加法，80x86 CPU 是如何进行的呢？早期的 CPU 只有字节大小的整型加法指令，那么它们是如何处理 16 位或 32 位数的加法运算呢？这主要是因为除了常规的加法指令外，CPU 还有带有进位的加法指令。当前的进位标志的值加上正常相加的和，其他就都和常规加法指令一样。如果要进行大的整型数据的加法，就把它们分成 CPU 能处理的大小。在最右侧的部分，使用普通的加法指令，得到低位部分相加的和。然后用加法指令进行下一部分的相加。此过程非常类似于大多数学生学习的十进制数字的加法过程，如果有必要，可以执行任意多个部分的相加，因为带进位的加法指令像普通的加法指令一样设置进位标志。大数字的减法运算过程是类似的，也要使用带借位的减法指令。

练习 3.2

1. 给出每条指令的 80x86 操作码和目标代码的字节总数。

a.	add	ax,Value	b.	sub	Value,ax
c.	sub	eax,10	d.	add	Double,10
e.	add	eax,[ebx]	f.	sub	[ebx],eax
g.	sub	dl,ch	h.	add	bl,5
i.	inc	bx	j.	dec	al
k.	dec	Double	l.	inc	BYTE PTR [esi]
m.	neg	eax	n.	neg	bh
o.	neg	Double	p.	neg	WORD PTR [ebx]

2. 对于这个程序的每个部分，假设已有指令执行“前”的值，给出该指令执行“后”的值。

	指令执行之前	执行指令	指令执行之后
a.	EBX: FF FF FF 75 ECX: 00 00 01 A2	add ebx,ecx	EBX, ECX, SF, ZF, CF, OF
b.	EBX: FF FF FF 75 ECX: 00 00 01 A2	sub ebx,ecx	EBX, ECX, SF, ZF, CF, OF
c.	BX: FF 75 CX: 01 A2	sub cx,bx	BX, CX, SF, ZF, CF, OF
d.	DX: 01 4B	add dx,40h	DX, SF, ZF, CF, OF
e.	EAX: 00 00 00 64	sub eax,100	EAX, SF, ZF, CF, OF
f.	AX: 0A 20 Value 中的字 :FF 20	add ax,Value	Value 中的字， AX, SF, ZF, CF, OF
g.	AX: 0A 20 Value 中的字 :FF 20	sub Value,ax	Value 中的字， AX, SF, ZF, CF, OF
h.	CX: 03 1A	inc cx	CX, SF, ZF
i.	EAX: 00 00 00 01	dec eax	EAX, SF, ZF
j.	Count 中的字 :00 99	inc Count	Count 中的字，SF，ZF
k.	Count 中的字 :00 99	dec count	Count 中的字，SF，ZF
l.	EBX: FF FF FF FF	neg ebx	EBX, SF, ZF
m.	CL: 5F	neg cl	CL, SF, ZF
n.	Value 中的字 :FB 3C	neg Value	Value 中的字 :SF，ZF

编程练习 3.2

1. 编写完整的汇编程序，计算表达式 x−2y+4z，其中 x、y、z 都是存储器中的双字，计算结果存放在 EAX 中。选择当前的月（1 ～ 12）、日（1 ～ 31）、年（写出 4 位数的年，如 2008）作为 x、y、z 的值，在程序汇编之前预测结果，并在 Windbg 中执行该程序。（提示：对于 4*z 不必用乘法指令实现。）
2. 编写完整的汇编程序，计算表达式 2（−a+b−1）+c，其中 a、b、c 是存储器中的双字，结果存放在 EAX 中。选择当地的区号作为 a 的值，一个电话号码的前三位数字为 b 的值，这个电话号码的后四位为 c 的值，在程序汇编之前预测结果，并在 Windbg 中执行该程序。
3. 编写完整的汇编程序，计算矩形的周长（2*length+2*width），其中长和宽都是存储器中的双字，结果存放在 EAX 中。选择一个壁球场的尺寸作为长和宽，在程序汇编之前预测结果，并在 Windbg 中执行该程序。（提示：可以在美国壁球网站上找到壁球场尺寸。）

3.3 乘法指令

80x86 体系结构有两条乘法指令，imul 指令把操作数作为有符号数，乘积结果的符号由有符号数的乘法规则决定。mul 指令用来处理无符号二进制数的乘法，乘积结果也是无符号的。如果是非负数进行乘法运算，通常使用 mul 而不是 imul，这是由于 mul 的速度较快一些。

mul 比 imul 指令简单，所以首先介绍 mul。mul 只有一个操作数：它的格式是

```
mul   source
```

源操作数（source）可以是字节、字或双字，而且它也可以放在存储器或寄存器中。另外，被乘数总是放在累加器中：如果是字节源操作数，则放在 AL 中；如果是字源操作数，则放在

AX 中；如果是双字源操作数，则放在 EAX 中。如果源操作数是字节操作数，那么它将和 AL 中的字节相乘，其结果是 16 位长，存放在 AX 寄存器中；如果源操作数是字操作数，它将和 AX 中的字相乘，结果是 32 位长，其中低 16 位存放在 AX 寄存器中，高 16 位存放在 DX 中；如果源操作数是双字操作数，那么它将和 EAX 中的双字相乘，其乘积结果是 64 位长，其中低 32 位存放在 EAX 中，高 32 位存放在 EDX 中。对于字节乘法，AX 中原有的值由乘积替换掉；对于字乘法，AX 和 DX 中的原始值被乘积替换掉；类似地，对于双字乘法，EAX 和 EDX 中的值将被计算乘积结果取代。在以上各种情况下，源操作数中的值都不会改变，除非它是目标寄存器长度的一半。

乘积长度是乘数和被乘数的两倍，似乎有些奇怪。但是，在一般的十进制乘法中，这种情况也可能出现。例如，两个 4 位数相乘，结果可能是 7 位或 8 位数。计算机做乘法操作时，为了避免目的地址空间太小，所以，乘积结果以两倍操作数的空间存放。

为什么 80x86 要将一个 32 位的乘积结果存放在 DX 和 AX 中，而不是寄存器 EAX 中呢？这是因为，mul 指令在 8086、8088、80186、80286 处理器中都是使用的 16 位寄存器，80386 引进了 32 位寄存器来扩展早期的计算机。对于 80486、奔腾以及以后的处理器也继续兼容早期的设计。

即使为乘积提供双倍长的存储空间，判断乘积是否与源操作数长度一致也是十分有用的，也就是说，乘积的高位部分是 0。使用 mul 指令，如果乘积的高位部分不为零，那么进位标志位 CF 和溢出标志位 OF 将置为 1，否则就置为 0。乘法指令执行后仅影响 AF、PF、SF 和 ZF 标志位，它们的值会重新设置。第 4 章（分支和循环）将提到一些检查标志位值的指令，根据标志位的值，乘积结果的高位部分可被安全地忽略掉。

图 3-14 总结了 mul 指令允许的操作数类型。注意，mul 不允许有立即操作数。

操作数	字节数	操作码
8 位寄存器	2	F6
16 位寄存器	2	F7
32 位寄存器	2	F7
存储器字节	21	F6
存储器字	21	F7
存储器双字	21	F7

图 3-14 mul 指令

下面的例子详细说明了乘法指令是如何执行的。

示例

指令执行之前	执行指令	指令执行之后				
EAX: 00 00 00 05	mul ebx	EDX	00	00	00	00
EBX: 00 00 00 02						
EDX: ?? ?? ?? ??		EAX	00	00	00	0A

		CF, OF 0
EAX: ?? ?? 00 05 EBX: ?? ?? 00 02 EDX: ?? ?? ?? ??	`mul bx`	EDX [?? ?? 00 00] EAX [?? ?? 00 0A]
		CF, OF 0
EAX: 00 00 00 0A EDX: ?? ?? ?? ??	`mul eax`	EDX [00 00 00 00] EAX [00 00 00 64]
		CF, OF 0
AX: ?? 05 Factor 中的字节 :FF	`mul Factor`	AX [04 FB]
		CF, OF 1

第一个例子给出了 EAX 和 EBX 中双字的乘法。EDX 的内容在乘法运算中没有用到，但是它被 64 位乘积结果 000000000000000A 的高 32 位所取代。进位标志位和溢出标志位全部清零，因为 EDX 中的值为 00000000。第二个例子除了操作数是单字长以外与第一个例子是相同的。DX 的内容在乘法运算中没有被用到，但是它被 32 位乘积结果 0000000A 的高 16 位所取代。进位标志位和溢出标志位全部清零，因为 DX 中的值为 0000。EAX 和 EDX 的高 16 位都没有改变。第三个例子是 EAX 和自身做乘法，它说明了，对于乘法运算来说，显示的源操作数可以与另一个类型相同的隐式乘数相乘。最后一个例子是 AL 中字节的乘法，另外一个乘数是存储器中一个字节 Factor，值等同于无符号的十进制数 255，得到乘积是 16 位无符号数 04FB，由于高位部分不为零，所以 CF 和 OF 同时置为 1。

有符号的乘法指令的助记符是是 `imul`。它有三种格式，每一种都有不同的操作数。第一种格式是：

```
imul    source
```

这种格式和 mul 一样，用源操作数（source）中的内容作为一个乘数，累加器作为另一个乘数，源操作数不能是立即数。根据源操作数大小，目标寄存器可能是 AX、DX:AX 或 EDX:EAX。如果高位字节是有意义的，进位和溢出标志位置为 1，否则置为 0。如果乘积是负数，高位部分全为 1，但如果低位的符号位也是 1，这就不重要了。类似地，低位的符号位为 1，那么所有的高位都为 0 就很重要了。如果整个双字长乘积表示不同的有符号数，而不是低位部分，那么 CF 和 OF 都置 1。

图 3-15 中总结了单操作数的 `imul` 指令。注意，这个图与图 3-14 是相同的。尽管 `mul` 和单操作数的 `imul` 指令的操作码是相同的，但两条指令的 mod-reg-r/m 字节的 reg 字段是有区别的。图 3-8 显示 `mul` 指令的 reg = 100，而 `imul` 指令的 reg = 101。

`imul` 的第二种格式是：

```
imul    register, source
```

操作数	字节数	操作码
8 位寄存器	2	F6
16 位寄存器	2	F7
32 位寄存器	2	F7
存储器字节	2+	F6
存储器字	2+	F7
存储器双字	2+	F7

图 3-15 imul 指令（单操作数格式）

在这种格式中，源操作数（source）可以在寄存器中、存储器中或是立即数，另一个乘数（register）在寄存器中，它也作为目标地址。操作数必须是字或双字，而不能是字节，乘积必须与乘数的长度一致，如果是这种情况，CF 和 OF 清零，否则都置 1。

图 3-16 介绍了两个操作数的 imul 指令。注意，某些指令具有两个字节长度的操作码。立即数操作数可以是目标寄存器的大小或者是一个字节长。在乘法运算前，单字节操作数根据符号位扩展——也就是说起始位和符号位相同，扩展后 16 位或 32 位的值表示原来同样的 8 位有符号操作数。

操作数 1	操作数 2	字节数	操作码
16 位寄存器	16 位寄存器	3	0FAF
32 位寄存器	32 位寄存器	3	0FAF
16 位寄存器	存储器字	3+	0FAF
32 位寄存器	存储器双字	3+	0FAF
16 位寄存器	字节立即数	3	6B
16 位寄存器	字立即数	4	69
32 位寄存器	字节立即数	3	6B
32 位寄存器	双字立即数	6	69

图 3-16 imul 指令（双操作数格式）

第三种格式是：

```
imul  register, source, immediate
```

在这个指令格式中，第一个操作数是寄存器，仅用来存放乘积，两个乘数一个可能在寄存器中或者在源操作数 source 指定的存储器中，另外一个是立即数。寄存器操作数和存储器操作数长度相同，都是 16 位或都是 32 位。如果乘积和目标寄存器长度一致，那么 CF 和 OF 清零；否则置 1。图 3-17 总结了三个操作数的 imul 指令。

下面的例子有助于理解 imul 指令。

寄存器目的操作数	源操作数	立即数操作数	字节数	操作码
16 位寄存器	16 位寄存器	字节	3	6B
16 位寄存器	16 位存储器	字	4	69
16 位寄存器	16 位存储器	字节	3+	6B
16 位寄存器	16 位存储器	字	4+	69
32 位寄存器	32 位寄存器	字节	3	6B
32 位寄存器	32 位寄存器	双字	6	69
32 位寄存器	32 位存储器	字节	3+	6B
32 位寄存器	32 位存储器	双字	6+	69

图 3-17 imul 指令（三个操作数格式）

示例

指令执行之前	执行指令	指令执行之后
EAX: 00 00 00 05 EBX: 00 00 00 02 EDX: ?? ?? ?? ??	`imul ebx`	EDX 00 00 00 00 EAX 00 00 00 0A CF, OF 0
EAX: ?? ?? 00 05 EBX: ?? ?? 00 02 EDX: ?? ?? ?? ??	`imul bx`	EDX ?? ?? 00 00 EAX ?? ?? 00 0A CF, OF 0
AX: ?? 05 Factor 中的字节：FF	`imul Factor`	AX FF FB CF, OF 0
EBX: 00 00 00 0A	`imul ebx,10`	EBX 00 00 00 64 CF, OF 0
ECX: FF FF FF F4 Double 中的双字： FF FF FF B2	`imul ecx, Double`	ECX 00 00 03 A8 CF, OF 0
Value 中的字 :08 F2 BX: ?? ??	`imul bx, Value, 1000`	BX F1 50 CF, OF 1

前面三个例子是单操作数格式，乘积长度是操作数长度的两倍。第一个例子给出了在 EAX 中的双字（这是隐含的操作数）和 EBX 中的数相乘，乘积结果存放在 EDX：EAX 中。第二个例子是 AX 和 BX 中单字长的值相乘，乘积结果存放在 DX：AX 中。第三个例子说明了 AL 中的 5 与 Factor 指示的存储器中的值（值为 −1）相乘，得到一个字长的乘积，其值为 −5，存于

AX 中。第四个例子给出了双操作数格式， EBX 中的 10 和立即数 10 相乘，得到的乘积 100 存放于 EBX 中。EDX 寄存器不用于两个操作数的格式，除非其中的一个乘数被指定放在 EDX 中。从第五个到最后一个例子都是两个负数相乘，得到的乘积为正。最后一个例子是三个操作数格式，十六进制数 8F2 与十进制数 1000 相乘，其结果是十六进制数的 22F150，因为值太大，不能存放于 BX 中，所以 CF 和 OF 全被置为 1，表明结果太大，低位部分存放于 BX 中。

早期的 80x86 CPU 处理器的每条指令都有固定的时钟周期数。随着生产的发展，对于现在的处理器，相同的指令要比原来的处理器耗费更少的时钟周期。随着时钟速率的提高，新的计算机拥有更快的速度。现在 80x86 处理器利用管道技术，比一次执行一条指令的效率要高。因此，很难确定所有指令的准确时间。但是，一般的乘法指令是 80x86 中执行速度最慢的指令。例如，要想计算 2 乘以 EAX 中的值，更有效的是用

```
add  eax, eax        ; double the value
```

而不是

```
imul  eax, 2        ; double the value
```

因此，无论是用汇编语言还是用一种高级语言编写程序时，用简单的加法就可以完成的工作要尽量避免运用乘法。

正如在本节中所看到的，80x86 体系结构有三种格式的乘法指令。值得注意的是，用来存放乘积结果的目的操作数不能是存储器操作数。这可能有些受限制，但是某些处理器的限制条件更多。实际上，大多数 8 位微处理器，包括 Intel 8080，都没有乘法指令；所有的乘法运算都必须通过软件编程来实现。

本节用一个程序例子作总结。该程序利用存储器中存放的矩形的长和宽，计算矩形的面积（长 * 宽）。（很明显，相比用汇编语言或者其他语言编写计算机程序来计算，使用一个计算器更合适。）图 3-18 给出了程序的源代码。注意，由于长和宽都是正数，所以该程序使用 mul

```
; program to find the area of a rectangle
; author:  R. Detmer
; date:  revised 5/2005

.386
.MODEL FLAT

ExitProcess PROTO NEAR32 stdcall, dwExitCode:DWORD

.STACK  4096              ; reserve 4096-byte stack

.DATA                     ; reserve storage for data
long        DWORD  35
wide        DWORD  27

.CODE                                 ; start of main program
                                      ; code
_start:
            mov    eax, long          ; length
            mul    wide               ; length * width

            INVOKE ExitProcess, 0     ; exit with return code 0
PUBLIC _start                         ; make entry point public
            END
```

图 3-18　计算矩形面积程序

而不是 imul 来计算乘积。在程序中，变量的长度和宽度命名为 long 和 wide，而不是 length 和 width，其原因是 length 和 width 是 MASM 的保留字。如果在汇编程序时，出现了一个不常见的错误，请查看附录 C，检查一下是否程序不小心用保留字作为标识符了。

图 3-19 中展示的是，在我们的计算矩形区域面积的程序中的 mul 指令执行后的 Windbg 的屏幕截图。十六进制数 3B1 在 EAX 中，但是应当注意到 EDX 中的数全被置零——实际上 mul 指令的计算结果是 00000000000003B1。

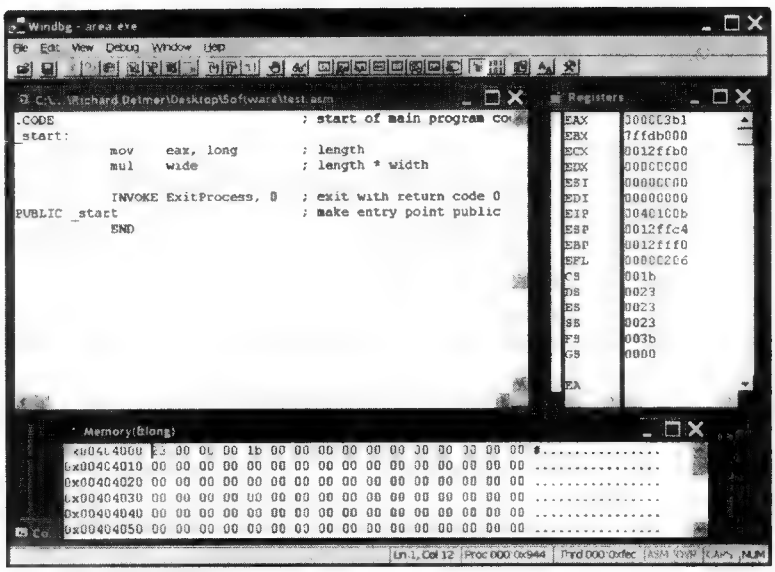


图 3-19 计算矩形面积程序的执行

练习 3.3

1. 对于如下问题，假设已有指令执行“前”的值，给出该指令执行“后”的值。

	指令执行之前	执行指令	指令执行之后
a.	EAX: FF FF FF E4 EBX: 00 00 00 02	mul ebx	EAX, EDX, CF, OF
b.	AX: FF FF FF E4 Value 中的双字： FF FF FF 3A	mul Value	EAX, EDX, CF, OF
c.	AX: FF FF	mul ax	AX, DX, CF, OF
d.	AL: 0F BH: 4C	mul bh	AX, CF, OF
e.	AL: F0 BH: C4	mul bh	AX, CF, OF
f.	EAX: 00 00 00 17 ECX: 00 00 00 B2	imul ecx	EAX, EDX, CF, OF
g.	EAX: FF FF FF E4 EBX: 00 00 04 C2	imul ebx	EAX, EDX, CF, OF
h.	AX: FF FF FF E4 Value 中的双字：	imul Value	EAX, EDX, CF, OF

	FF FF FF 3A		
i.	EAX: FF FF FF FF	imul eax	EAX, EDX, CF, OF
j.	AL: 0F		
	BH: 4C	imul bh	AX, CF, OF
k.	AL: F0		
	BH: C4	imul bh	AX, CF, OF

2. 给出习题 1 中每条指令的操作码。
3. 对于如下问题，假设已有指令执行“前”的值，给出该指令执行“后”的值。

	指令执行之前	执行指令	指令执行之后
a.	EBX: 00 00 00 17		
	ECX: 00 00 00 B2	imul ebx, ecx	EBX, CF, OF
b.	EAX: FF FF FF E4		
	EBX: 00 00 04 C2	imul eax, ebx	EAX, CF, OF
c.	EAX: 00 00 0F B2	imul eax, 15	EAX, CF, OF
d.	ECX: 00 00 7C E4		
	Mult 中的双字：	imul ecx, Mult	ECX, CF, OF
	00 00 65 ED		
e.	DX: 7C E4		
	BX: 49 30	imul dx, bx	DX, CF, OF
f.	DX: 0F E4		
	Value中的字：04 C2	imul dx, Value	DX, CF, OF
g.	EBX: 00 00 04 C2	imul ebx, -10	EBX, CF, OF
h.	ECX: FF FF FF E4	imul ebx, ecx, 5	EBX, CF, OF
i.	EDX: 00 00 00 64	imul eax, edx, 10	EAX, CF, OF

4. 给出习题 3 中每条指令的操作码。
5. 假定对于 x 的某个值，需要计算下列多项式的值：

   $p(x)=5x^3-7x^2+3x-10$

   如果以较直截的方法计算该多项式的值，如：

   5*x*x*x−7*x*x+3*x−10

   则有 6 次乘法运算和 3 次加 / 减法运算。基于 Horner 方法计算该多项式的值，则等价于如下形式：

   ((5*x−7)*x+3)*x−10

   这个表达式中仅有三次乘法运算。

   假定 x 的值存放在 EAX 寄存器中。

   a. 根据第二种表达方式编写 80x86 汇编程序，计算 p(x) 多项式的值，并将计算结果存放在 EAX 中。

   b. 根据第三种表达方式编写 80x86 汇编程序，计算 p(x) 多项式的值，并将计算结果存放在 EAX 中。

   c. 比较以上两种方法需要的目标代码的字节总数。
6. 80x86 体系结构对于有符号数和无符号数的乘法有不同的指令，而对于有符号数和无符号数的加法却没有各自的指令。为什么对于乘法需要有不同的指令，而对于加法却

没有呢？

编程练习 3.3

1. 假设图 3-18 中的计算矩形面积的程序的长为 30 000，宽为 20 000。编译并运行该程序。在 Windbg 中，执行 mul 指令后，矩形面积的结果是多少？
2. 编写一个完整的 80x86 汇编语言程序，箱子的长、宽、高存放在存储器中，计算箱子的体积（长 × 宽 × 高），结果存放在 EAX 中。
3. 编写一个完整的 80x86 汇编程序，箱子的长、宽、高存放在存储器中，计算箱子的表面积 2×（长 × 宽+长 × 高+宽 × 高）。
4. 假定某人有一定数量的硬币（便士、五分、一角、二角五分、五十分和一美元硬币），如果要计算总的硬币值（用分作单位）和总的硬币个数。编写一个完整的 80x86 汇编程序，计算总的硬币值，存放在 EAX；计算总的硬币个数，存放在 ECX 中。

3.4 除法指令

Intel 80x86 的除法指令和单操作数乘法指令很相似，指令 idiv 用于有符号二进制补码整数的除法，指令 div 用于无符号整数的除法。回想一下，单操作数乘法指令用乘数和被乘数相乘，并得到一个两倍长的乘积。而除法指令用一个两倍长的数作为被除数，用一个单倍长度的数作为除数，最后得到单倍长度的商和单倍长度的余数。在开始除法运算之前，80x86 用指令来产生一个双倍长的被除数。

除法指令的格式是

```
idiv    source
```

和

```
div     source
```

源操作数（source）就是除数。除数可以存放在寄存器或存储器中，但不能是立即数。idiv 和 div 使有隐含的被除数。如果源操作数是字节长度的，那么双字节长度的被除数也就是一个字长，必须存储在寄存器 AX 中。如果源操作数是一个字长，那么双倍长度的被除数是双字长度，存放在 DX:AX 中，即它的低 16 位存放在 AX 寄存器中，高 16 位在 DX 寄存器中；如果源操作数是一个双字长的，那么被除数应是四字长（64 位），存放在 EDX:EAX 中，即它的低 32 位存放在 EAX 寄存器，高 32 位存放在 EDX 寄存器中。

图 3-20 总结了 80x86 除法指令中被除数、除数、商和余数的各种情况。

源操作数（除数）大小	隐含的操作数（被除数）	商	余数
字节	AX	AL	AH
字	DX:AX	AX	DX
双字	EDX:EAX	EAX	EDX

图 3-20 80x86 除法指令的操作数和结果

除法指令不会改变源操作数（除数）。AX 中字长的被除数，被字节长的除数进行除操作后，商将放入寄存器 AL 中，余数将放入寄存器 AH 中；DX 和 AX 中的一个双字被字长的除数进行除操作后，商将存入 AX 寄存器中，而余数存入寄存器 DX 中；EDX 和 EAX 中的一个四字长的数被双字长的除数进行除操作后，商将存入 EAX 寄存器，而余数将存入 EDX 寄存器。对于所有的除法运算，被除数、除数、商、余数必须满足下面的等式

被除数＝商 * 除数＋余数

对于无符号的 div 操作，被除数、除数、商和余数都被当作非负数看待。对于有符号数除法指令 idiv，商的符号使用一般的符号规则，由被除数和除数决定，余数的符号总是和被除数的相同。

除法指令不对任何标志位赋值。它们可能会改变 AF、CF、OF、PF、SF 和 ZF 预先设置的值。下面的例子说明了除法指令是如何执行的。

示例

指令执行之前	执行指令	指令执行之后
EDX: 00 00 00 00 EAX: 00 00 00 64 EBX: 00 00 00 0D	`div ebx ; 100/13`	EDX: 00 00 00 09 EAX: 00 00 00 07
DX: 00 00 AX: 00 64 CX: 00 0D	`idiv cx ; 100/13`	DX: 00 09 AX: 00 07
AX: 00 64 Divisor中的字节 :0D	`div Divisor ; 100/13`	AX: 09 07

在这些例子中，十进制数 100 被 13 除。因为

100 = 7*13 + 9

商是 7，余数是 9。对于双字长的除数，商是在 EAX 中，余数在 EDX。对于单字长的除数，商是在 AX 中，余数在 DX 中。对于单字节长的除数，商是在 AL 中，余数在 AH 中。

对于除法运算，若其中被除数或除数为负，类似于上面的等式

100 = (−7)*(−13) + 9　　（100/−13，商 −7，余数 9）

−100 = (−7)*13 + (−9)　　（−100/13，商 −7，余数 −9）

−100 = 7*(−13) + (−9)　　（−100/−13，商 7，余数 −9）

注意，在这些例子中，可以发现余数的符号与被除数的符号是相同的。接下来的例子给出了对双字长除数 13 或 −13 的等式。其中，在第二个和第三个例子中，被除数 −100 在 EDX 和 EAX 寄存器中被表示成 64 位数 FF FF FF FF FF FF FF 9C。

示例

指令执行之前	执行指令	指令执行之后				
EDX: 00 00 00 00 EAX: 00 00 00 64 ECX: FF FF FF F3	`idiv ecx ; 100/(-13)`	EDX	00	00	00	09
		EAX	FF	FF	FF	F9
EDX: FF FF FF FF EAX: FF FF FF 9C ECX: 00 00 00 0D	`idiv ecx ; -100/13`	EDX	FF	FF	FF	F7
		EAX	FF	FF	FF	F9
EDX: FF FF FF FF EAX: FF FF FF 9C ECX: FF FF FF F3	`idiv ecx ; -100/(-13)`	EDX	FF	FF	FF	F7
		EAX	00	00	00	07

最后，这两个例子有助于说明有符号数除法和无符号数除法的不同之处。使用有符号数的除法，−511 除以 −32，得到商 15 和余数 −31。对于无符号数除法，65025 除以 255，得到商 255 和余数 0。

示例

指令执行之前	执行指令	指令执行之后		
AX: FE 01 BL: E0	`idiv bl ; -511/(-32)`	AX	E1	0F
AX: FE 01 BL: FF	`div bl ; 65025/255`	AX	00	FF

对于乘法，在每个单操作数格式中，双倍长的目标地址能确保存放乘积——这样在单操作数乘法运算中就不会出错。但是，在除法中就有可能有错，一个显而易见的错误，就是除数为 0 的情况。还有可能出现错误的情况是，是商太大而不能存入单倍长的目标地址中。比如，00 02 46 8A 除以 00 02，商 12345 太大而不能存入 AX 寄存器中。如果在除法操作运算时出错，80x86 将产生异常（exception）。对于不同的系统来说，处理异常的程序或者中断控制器是各种各样的。在 Windbg 中，如果程序运行时出现了除法的错误，程序就会中断。

图 3-21 列出了 div 和 idiv 指令所允许的操作数类型。这些指令的操作码相同，正如图 3-8 所示，这些指令也是利用 mod-reg-r/m 字节中的 reg 字段来区别。

对给定长度的操作数进行算术运算时，在执行除法操作前，被除数必须转变成两倍长度的数。对于无符号数的除法，一个双字长大小的被除数必须转变成四倍字长大小的数，这个数存放在 EDX 寄存器中，它的高位部分都是零。实现的方法有很多种，下面是其中的两种方法。

```
mov     edx, 0
```

操作数	字节数	操作码
8 位寄存器	2	F6
16 位寄存器	2	F7
32 位寄存器	2	F7
存储器字节	2+	F6
存储器字	2+	F7
存储器双字	2+	F7

图 3-21 div 和 idiv 指令

和

```
sub     edx, edx
```

在做无符号数除法前，如果被除数是字操作数，可以用类似的指令将 DX 的高位置 0，如果被除数是字节操作数，则可将 AH 的高位置 0。

对于有符号数除法，情况就要复杂多了。正的被除数高位必须用 0 扩展，负的被除数高位必须用 1 扩展。80x86 有三个指令可以实现这项工作，cbw、cwd 以及 cdq 指令。与前面所提到的指令不同，这三条指令都没有操作数。其中 cbw 指令是用 AL 作为源操作数，AX 作为目的操作数；cwd 用 AX 作为源操作数，DX：AX 作为目的操作数；cdq 是用 EAX 作为源操作数，EDX：EAX 作为目的操作数。源寄存器的值不会改变，但它们会作为有符号数扩展到 AH、DX 或者 EDX 中。这些指令总结在图 3-22 中。

指令	字节数	操作码
cbw	1	98
cwd	1	99
cdq	1	99

图 3-22 cbw、cwd 和 cdq 指令

cbw 指令（将字节转换为字）将 AL 寄存器中的二进制补码数扩展为 AX 中的一个字长的数。cwd（将字转换为双字）指令将 AX 寄存器中的单字长的二进制数扩展成双字长数，存放在寄存器 DX 和 AX 中。cdq（将双字转换为四字）将 EAX 中的双字扩展为四字长，放在寄存器 EDX 和 EAX 中。每条指令都复制源数据的符号位到运算结果的高位部分中的每一位，这些指令都不影响标志位。下面是一些例子。

示例

指令执行之前	执行指令	指令执行之后				
EAX: FF FF FA 13 EDX: ?? ?? ?? ??	cdq	EDX	FF	FF	FF	FF
		EAX	FF	FF	FF	13

EAX: FF FF FA 13 EDX: ?? ?? ?? ??	cdq	EDX	FF	FF	FF	FF
		EAX	FF	FF	FF	13
AX: 07 0D DX: ?? ??	cwd	DX	00	00		
		AX	07	0D		
AX: ?? 53	cbw	AX	00	53		
AX: ?? C6	cbw					
		AX	FF	C6		

本小节用一个简单的程序作总结，该程序是将摄氏温度转换成华氏温度。图 3-23 给出了源代码。实现的方程是

$F=(9/5)*C+32$

其中 F 是华氏温度，C 是摄氏温度。程序中摄氏温度的初始值为 35，这个值是一个双字长的数，存储在 cTemp 存储器中，得到的最终值是相应的华氏温度，它也是一个双字长的数，存储在 fTemp 存储器中。初始值现在用的是 35，当然，也可用其他值作为初始值。

目前所涉及的算术指令都是整数运算，程序得到的结果是对小数部分进行取整的结果。在除以 5 之前，将 9 和 cTemp 相乘，这一点非常重要，因为 9/5 的整数商为 1。如果 cTemp 先被 5 除，

```
; program to convert Celsius temperature to Fahrenheit
; uses formula  F = (9/5)*C + 32
; author:  R. Detmer
; date:  revised 5/05

.386
.MODEL FLAT

ExitProcess PROTO NEAR32 stdcall, dwExitCode:DWORD
.STACK  4096              ; reserve 4096-byte stack

.DATA                     ; reserve storage for data
cTemp       DWORD  35     ; Celsius temperature
fTemp       DWORD  ?      ; Fahrenheit temperature

.CODE
_start:
            mov    eax, cTemp        ; start with Celsius temperature
            imul   eax,9             ; C*9
            add    eax,2             ; rounding factor for division
            mov    ebx,5             ; divisor
            cdq                      ; prepare for division
            idiv   ebx               ; C*9/5
            add    eax,32            ; C*9/5 + 32
            mov    fTemp, eax        ; save result

            INVOKE ExitProcess, 0    ; exit with return code 0
PUBLIC _start                        ; make entry point public
            END
```

图 3-23 转换摄氏温度为华氏温度程序

再与 9 相乘，则这种方法比第一种方法产生的误差更大。（为什么？）为了得到取整运算的正确结果，在除之前将除数的一半加到被除数中。由于方程中的除数是 5，取舍后的 2 加到被除数中。注意，cwd 指令在做除法前通常扩展被除数。

图 3-24 展示的是 Windbg 中该程序的结束部分。如果手算，35*9 + 2 结果是 317。317 除以 5，商是 63，余数 2（仍然在 EDX 中）。最后，用 32 加 63 得到 95，在 EAX 和第二个存储器中显示为双字长的十六进制数 5F。

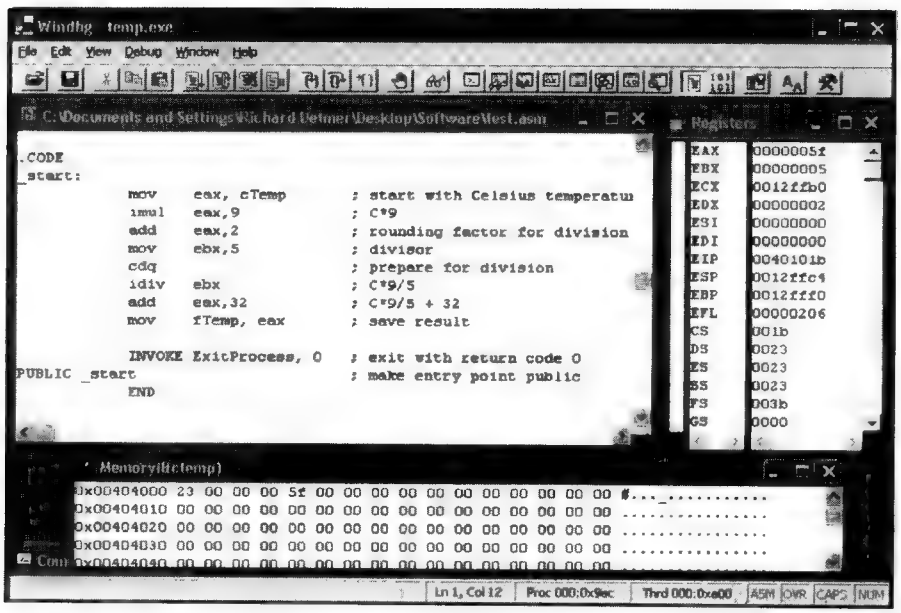


图 3-24 摄氏温度转换为华氏温度程序的执行

练习 3.4

1. 对于如下问题的每部分，假设已有指令执行“前”的值，给出该指令执行“后”的值。这些指令中有些指令会造成除法运算出错，指出这样的指令。

	指令执行之前	执行指令	指令执行之后
a.	EDX: 00 00 00 00 EAX: 00 00 00 9A EBX: 00 00 00 0F	idiv ebx	EDX, EAX
b.	AX: FF 75 Count 中的字节：FC	idiv Count	AX
c.	AX: FF 75 Count 中的字节：FC	div Count	AX
d.	DX: FF FF AX: FF 9A CX: 00 00	idiv cx	DX, AX
e.	EDX: FF FF FF FF EAX: FF FF FF 9A ECX: FF FF FF C7	idiv ecx	EDX, EAX

f. DX: 00 00
AX: 05 9A
CX: FF C7 `idiv cx` DX, AX

g. DX: 00 00
AX: 05 9A
CX: 00 00 `idiv cx` DX, AX

h. EDX: 00 00 00 00
EAX: 00 00 01 5D
EBX: 00 00 00 08 `idiv ebx` EDX, EAX

2. 给出练习 1 中每条指令的操作码。
3. 在做无符号数除法之前，本节给出了两种置 EDX 为 0 的方法。使用

```
mov  edx , 0
```

或

```
sub  edx ,edx
```

哪条指令所需的目标代码字节数更少？

编程练习 3.4

1. 将华氏温度转换为摄氏温度的公式为：
   $C = (5/9) * (F-32)$
   写一个完整的 80x86 汇编语言程序，该程序将华氏温度转换为对应的摄氏温度。
2. 写一个完整的 80x86 汇编语言程序，该程序有四个双字长的成绩：Grade1，Grade2，Grade3，Grade4。计算这四个成绩的总和（sum）以及四个成绩的平均分（sum/4），分别放在双字长的 Sum 和 Avg 存储单元中。
3. 写一个完整的 80x86 汇编语言程序，该程序有四个双字长的成绩：Grade1，Grade2，Grade3，Grade4。假设最后一个成绩是期末考试成绩，它与其他三个成绩一样计算，但是它要计算两次。显示成绩的总和 sum（最后一个成绩加了两次）和平均成绩（sum/5）。
4. 写一个完整的 80x86 汇编语言程序，该程序有四个双字长的成绩：Grade1，Grade2，Grade3，Grade4；以及 4 个权系数：Weight1，Weight2，Weight3，Weight4。每个权系数表示其对应的成绩应该在 sum 中计算的次数。加权总和为：
   WeightedSum=Grade1 * Weight1 + Grade2 * Weight2 + Grade3 * Weight3 + Grade4 * Weight4
   权系数总和为：SumOfWeights=Weight1+Weight2+Weight3+Weigt4
   计算加权后的总和、权系数的和以及加权后的平均值（WeightedSum / SumOfWeight）。将结果以双字形式放在存储器中。

3.5 本章小结

Intel 80x86 `mov` 指令用来将数据从一个位置复制到另一个位置。但并不是所有的源位置和目标位置的逻辑组合都是可以的。`xchg` 指令交换存储在两个地址中的数据。

80x86 体系结构有很多用于字节长、字长和双字长整数计算的指令集。`add` 和 `sub` 指令用来做加法和减法；`inc` 和 `dec` 分别做加 1 和减 1 运算。`neg` 指令对操作数进行相应的二进制

的取补。

有两个乘法和两个除法指令。imul 和 idiv 指令，假定它们的操作数是有符号的二进制补码数；mul 和 div 指令，假定它们的操作数是无符号的。许多乘法指令使用单倍长度的操作数，并产生一个双倍长度的乘积；其他格式产生一个和乘数相同长度的乘积。除法指令总是以一个双倍长度的被除数和一个单倍长度的除数开始；运算结果是一个单倍长度的商和一个单倍长度的余数。在做有符号数除法运算前，cbw、cwd、cdq 指令用于扩展被除数为双倍长度。乘法运算时，标志位的设置提示可能出现的错误；除法运算时发生的错误会产生一个硬件异常，该异常触发一个过程来处理错误。

操作数在寄存器中的指令通常比引用操作数在存储器中的指令执行速度要快。乘法和除法指令相对于加减法指令执行得要慢。

第 4 章　分支与循环

计算机强大的处理能力不仅在于它可选择性地执行代码，还在于它可高速地执行重复的算法。用高级语言如 Java 和 C++ 语言来编写程序，用 if-then、if-then-else 和 case 结构来选择性地执行代码，并且运用 while 循环（循环之前检查条件）、until 循环（循环之后检查条件）、for（计数器控制）来重复执行代码。一些高级语言对于无条件分支结构使用 goto 语句实现。更早期的一些语言（比如旧版本的 Basic 语言），依靠很简单的 if 语句和大量的 goto 语句，执行有选择的分支和循环结构。

用 80x86 汇编语言编程和以前用 Basic 语言编程相似。80x86 微处理器可以执行一些与 for 语句功能大致相同的语句。但是，对于大多数分支和循环结构，80x86 是用那些比 if 或 goto 语句更简单，甚至更原始的语句来完成。本章旨在论述 if-then、if-then-else、while、until 和 for 等语言结构在机器上是如何实现的。

4.1　无条件转移指令

80x86 jmp（跳转）指令与高级语言的 goto 语句类似，用汇编语言编写代码时，jmp 语句格式如下：

```
jmp   StatementLabel
```

其中 StatementLabel 标号与其他汇编语言程序语句中的名称字段一致。回想一下，当使用标号标识一个可执行语句时，名称字段后面是跟着冒号（：），但 jmp 语句中标号不用冒号。例如，在程序应该终止时如果有两个选择条件，则代码可以包含如下内容：

```
jmp     quit                    ; exit from program
          .
          .
quit:   INVOKE ExitProcess, 0   ; exit with return code 0
          .
          .
```

图 4-1 显示了一个完整的例子，即无限循环的程序。计算 1 + 2 + . . . + *n* 的 *n* 次循环。这个程序实现可用下面的伪代码设计：

```
number := 0;
sum := 0;
forever loop
      add 1 to number;
      add number to sum;
end loop;
```

编写码实现像这样的一个设计时要有计划地使用寄存器和存储器。在这个程序实现中，number 值存储在寄存器 EBX 中，并且 sum 值存储在寄存器 EAX 中。没有用存储器存放数据。

在 图 4-2 中，调试器准备开始执行这个程序。这里不是使用一次一条语句的单步方法来执行程序，而是在程序中设置**断点**。当程序开始运行时，它将会在设置断点语句的位置暂停。在这个例子中，断点设置在 jmp 指令处，以便能查看 sum（存放在寄存器 EAX 中）和 number（存放在寄存器 EBX 中）的内容，并进行下一次的循环。因此，单击 jmp 指令所在行的任何位置，并选择菜单栏中的“Edit/Breakpoints”。“Break at Location”应该被选中。点击“Ok”按钮，对话框将被关闭，并且 jmp 指令所在的行将以红色字体突出显示。现在，点击“Run to Cursor”

```
; program to find sum 1 + 2 + . . . + n for n = 1, 2, . . .
; author:  R. Detmer
; date:  7/2005

.386
.MODEL FLAT

.STACK  4096                ; reserve 4096-byte stack

.DATA                       ; reserve storage for data

.CODE                                   ; start of main program code
_start:
            mov     ebx,0               ; number := 0
            mov     eax,0               ; sum := 0

forever:    inc     ebx                 ; add 1 to number
            add     eax, ebx            ; add number to sum
            jmp     forever             ; repeat
PUBLIC _start                           ; make entry point public
            END
```

图 4-1　无限循环结构的程序

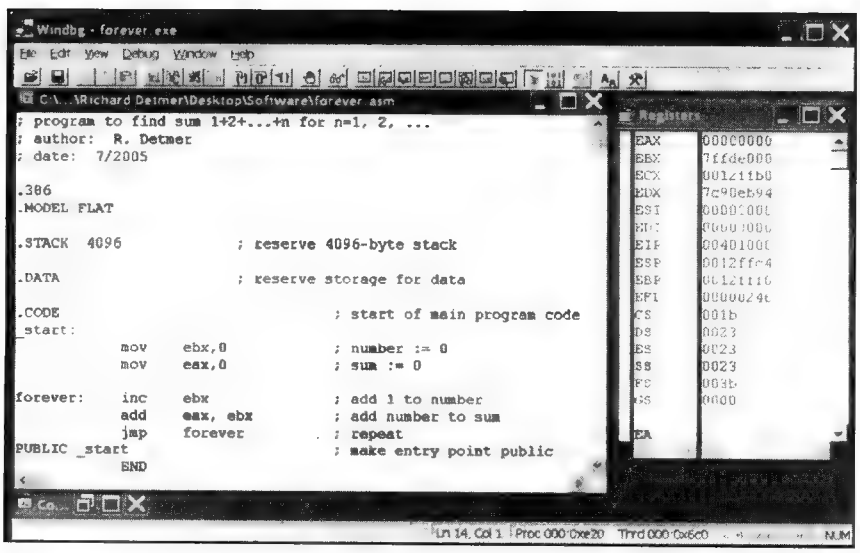


图 4-2　开始执行之前的程序

按钮，执行 jmp 指令后面的语句。图 4-3 显示了程序执行后的结果。注意，第一次到达断点时，sum 和 number 的值都是 1。

点击了 5 次“Run to Cursor”按钮后，最终的 Windbg 窗口如图 4-4 所示。总共运行 6 次循环迭代之后，number（存放在寄存器 EBX 中）的值为 6，而 sum（存放在寄存器 EBX 中）的值为 15_{16}（用十进制表示为 21_{10}），是 1 + 2+ 3 + 4 + 5 + 6 的正确结果。

图 4-1 程序中的一个 jmp 指令把控制转移到 这条 jmp 语句本身之前。这种方法称为**向后引用**（backward reference）。代码

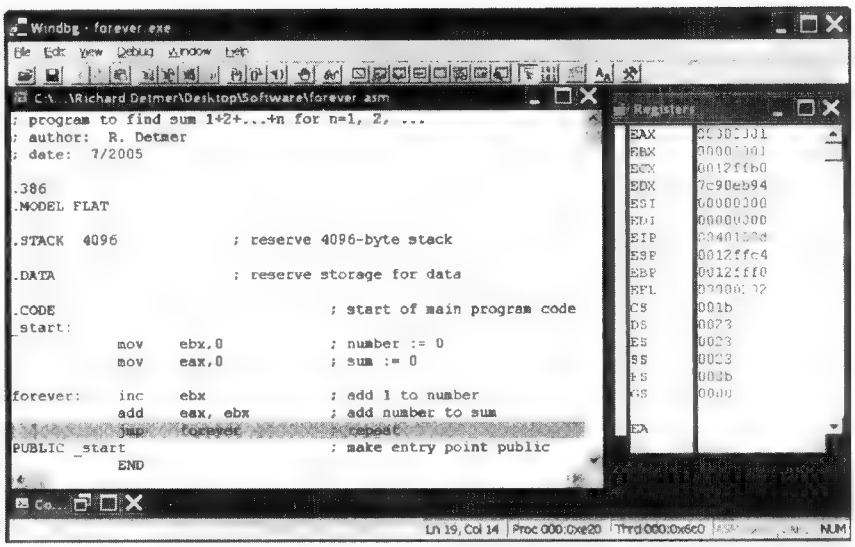


图 4-3 执行循环体的第 1 次迭代后的程序

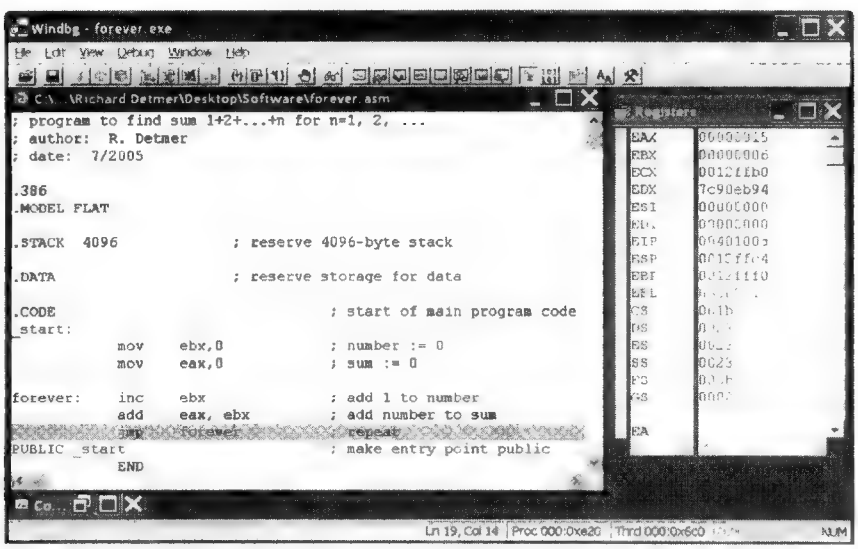


图 4-4 执行循环体的第 6 次迭代后的程序

```
jmp     quit                    ; exit from program
          .
          .
quit:  INVOKE ExitProcess, 0    ; exit with return code 0
```

说明的是一个**向前引用**（forward reference）。

80x86 的 jmp 指令主要分为两类：**段间转移**（intersegment）和**段内转移**（intrasegment）。它们都是通过改变指令指针寄存器（EIP）的值来实现的，这样下一个要执行指令的地址来自于新的地址，而不是当前指令的下一个地址。段间转移改变代码段寄存器 CS 和 EIP 内容。但是，在平面存储模式的编程中不会出现这种情况。因此，这里不讨论这些指令。图 4-5 列出了段内转移指令，其中，前面两条指令是最常用的。

类型	字节数	操作码
near 跳转	5	E9
short 跳转	2	EB
寄存器间接	2	FF
存储器间接	2+	FF

图 4-5　jmp 指令

每条相对转移指令包含从 jmp 指令本身开始的目标指令的位移量。这个位移量加在下一个指令的地址上就是要寻找的目标地址。位移量是一个有符号数，对于向前引用，它是正数；而对于向后引用，它是负数。对于相对短的转移指令，只存储一个单字节的位移量。在做加法运算之前，这个位移量首先扩展为双字。相对近的转移包含 32 位的位移量。

8 位位移量是一个相对较短（short）的转移，可以在 jmp 指令之前，转移到 128 字节以内的目标地址；或者在 jmp 指令之后，转移到 127 字节以内的目标地址。位移量是通过 jmp 本身的目标代码后面的字节数来计算的，这是因为在执行一条指令时，逻辑上 EIP 包含了下一条将要执行的指令的地址。32 位位移量是一个相对近（near）的转移，在 jmp 指令之前，可转移到 2 147 483 648 字节内的目标地址；或在 jmp 指令之后，转移到 2 147 483 647 字节以内的目标地址。

无论是相对短的还是近的转移指令，在编写代码时没有什么区别。为了让代码紧凑，如果目标地址范围不大，汇编器使用短的转移指令。如果目标地址超过 128 个字节，汇编器自动使用近的转移指令。

间接转移指令使用一个 32 位的地址作为目标地址，而不是用位移量；但是，这个地址并没有写在指令中，而是保存在寄存器或双字的存储器中。因此，格式

```
jmp     edx
```

意思是转移到存储于 EDX 中的地址处。存储器间接形式可以使用任一有效的双字内存地址。如果 Target 在数据项中声明为双字，那么

```
jmp     Target
```

将转移到存储于 Target 的双字地址处，而不是指数据项中的这个双字。用寄存器间接寻址，可以用

```
jmp     DWORD PTR  [ebx]
```

它将转移到存储在双字中的地址，而双字的地址是在 EBX 中！但是这些间接寻址形式都很少用。

练习 4.1

1. 如果执行下面的语句

```
hardLoop:    jmp     hardLoop
```

继续执行循环语句，这个语句的目标代码是什么？（如果不能推算出来，把这个语句放到程序中，编译程序并查看清单文件。）

2. 指出下列代码段中的每条 jmp 指令的类型（相对近转移，相对短转移，寄存器间接转移，或存储器间接转移）。

```
.DATA
          ...
addrStore DWORD  ?
          ...
.CODE
          ...
doAgain:
          ... (3 instructions)
          jmp  doAgain
          ... (200 instructions)
          jmp  doAgain
          ...
          jmp  addrStore
          ...
          jmp  eax
          ...
          jmp  [edi]
```

编程练习 4.1

修改图 4-1 中的程序，计算乘积，*n* 次循环后，EAX 中的值是 1*2*...**n* 的值。

4.2 条件转移指令、比较指令和 if 结构

在 80x86 机器语言中，条件转移指令可以实现 if 结构、其他选择结构以及循环结构。条件转移指令有很多，每种指令的格式如下：

```
j---     targetStatement
```

其中，助记符的最后部分定义了执行转移的条件。如果条件满足，则发生转移；否则执行下一条指令（条件转移后面的那一条指令）。

但有些条件转移指令的条件是由标志寄存器中的标志位来设置的，如 jcxz/jecxz 指令，

它们将在 4.4 节中介绍。例如，指令

```
jz          endWhile
```

如果零标志位被设置为 1，则转移到标号 endWhile 的语句；否则执行下面的语句（助记符 jz 表示“零转移”）。

条件转移指令不改变任何标志位，只根据已设置的标志位的值做出相应的动作。回想一下，标志寄存器中标志位是如何取值的。有些指令（如 mov）保持部分或全部的标志位不变，有些指令（如 add）根据结果的值来设置某些标志位，还有一些指令（如 div）对某些标志位的改变是不可预知的。因此，也就不能确定标志位的值。

例如，假设 EAX 寄存器中的值与一个表示账户余额的和相加，并且根据新的余额是否是正数、零或负数有三种不同的处理方法。其伪代码设计是：

```
add value to balance;

if balance < 0
then
    ... { design for negative balance }
elseif balance = 0
then
    ... { design for zero balance }
else
    ... { design for positive balance }
end if;
```

假设余额 balance 以双字形式存放存储器中，且 value 存储在 EAX 寄存器，下面是实现这个设计的 80x86 代码段：

```
            add   balance,eax       ; add value to balance
            jns   elseIfZero        ; jump if balance not negative
            ...                     ; code for negative balance
            jmp   endBalanceCheck
elseIfZero: jnz   elsePos           ; jump if balance not zero
            ...                     ; code for zero balance
            jmp   endBalanceCheck
elsePos:    ...                     ; code for positive balance

endBalanceCheck:
```

通过 add 指令来设置或清零相应的标志位。在上面的代码段中，其他的指令都不会更改标志位。这段代码首先检查 balance 是否小于 0（balance<0）。完成这项工作可以用下面的指令：

```
jns         elseIfZero
```

这条指令的意思是，如果符号标志位为 0，则转移到 elseIfZero；也就是说，如果（balance<0）不成立，则转移到 elseIfZero。跟在这条指令后的代码，与该伪代码设计中的第一个 then 后面的

语句是一致的，即，如果 balance<0 条件为真时所要执行的语句。语句

```
jmp     endBalanceCheck
```

是这个模块语句中的最后一个语句，它是必需的，这样 CPU 能跳过适合其他情况的语句。如果第一个条件跳转转移转到 elseIfZero，则余额 balance 必须是非负的（零或正数）。这样做的目的是检查 balance 的值是否为 0。如果零标志位 ZF 为 0，那么指令

```
elseIfZero:   jnz       elsePos
```

将跳转到 elsePos。最后一条设置标志位的指令是开始的 add 指令，这样如果余额 balance 不为 0 时发生转移。当 balance 为 0，程序必须无条件转移到 endBalanceCheck，这时程序再次结束。最后，程序中与 else 对应的代码在 elsePos 处。就像高级语言一样，第三个检查是不必要的，因为最后只剩下一种可能的情况，即 balance 为正。代码的最后部分不必用转移指令转到 endBalanceCheck，因为程序最终会执行到这里。

以上的 80x86 代码是直接按照程序中语句的顺序来执行的。如果用汇编语言编程，那么，较好的方法是：首先是仔细设计，有效地编码，然后进行检查，看看是否有必要修改，使之更加有效，许多高级语言编译器都是这样做的。很多机器语言程序与翻译过来的高级语言的语句的顺序是一致的。最后，编译程序优化代码，为了提高效率，再重新排列一些语句的顺序。

上面代码中，标号 endBalanceCheck 本身就占用一行。从技术上讲，这个标号将指向任何跟在它后面的语句的地址。但是，把它作为当前设计结构中的一部分，而不考虑接下来要做什么，这样更简单。因为，即使改变该结构后面的内容，这个结构的代码仍保持不变。如果下一条语句需要另外的标号，那也完全没有问题——多个标号仍能指向存储器中的同一位置。标号不是目标代码的一部分，因此多余的标号，并不会增加目标代码的长度，也不会增加运行时间。

当编写代码进行实际设计时，经常需要使用像 if、then、else 和 endif 这样的标号。但是，IF、ELSE 和 ENDIF 都是 MASM 的指令，因此它们不能被用作标号。除此之外，IF1、IF2 和其他一些可能想到的符号也是备用指令。一个解决办法是采用较长的描述性符号作为标号，如上例中的 elseIfZero。由于任何保留字都不能包含下划线，因此，还有一个解决办法是，当原程序中含有并列的关键字时，使用像 if_1 和 endif_2 为这样的符号作为标号。

术语**设置**（set）或**重置**（reset）分别是指对一个标志位置 1 或 0（有时使用 clear 代替 reset）。许多指令可用来为标志位置 1 或清 0，不过，使用 cmp（比较）指令为标志位赋值可能是最常用的方法。

每条 cmp 指令都对两个操作数进行比较，并为 AF、CF、OF、PF、SF 和 ZF 标志位置 0 或置 1。cmp 指令的唯一任务就是确定标志位的值，这不仅仅是其他功能的副作用。每条 cmp 指令的格式如下：

```
cmp         operand1,   operand2
```

cmp 指令通过计算 operand1 减去 operand2 的值来进行比较，就像一条 sub 指令。标志位的设置是由差值和执行减法时的情况来决定的。cmp 指令和 sub 指令的不同之处在于，cmp 指

令中，位于 operand1 的值不会改变。本书主要讨论的标志位是 CF、OF、SF 和 ZF。减法中有借位时，将进位标志位 CF 置 1；没有借位时，将其清 0。有溢出时，将溢出标志位 OF 置 1；否则清 0。如果差值是一个负的二进制补码数，则将符号标志位 SF 置 1，否则清 0。最后，如果差值为 0，则零标志位 ZF 置 1；如果不为 0，则清 0。

下面举例说明，对于一些常见的表示字节长度的数进行比较，标志位是如何设置的。回想一下，减法操作无论对于无符号数还是有符号数（二进制补码）来说，都是一样的。就像一个只有一位的数，它的形式即可以用无符号数表示，也可以用有符号数表示。但对于标志位而言，无论是对无符号数还是有符号数进行比较后，可能会有不同的解释。下表列出了有符号数和无符号数比较时操作数之间的关系。

	操作数 1（op1）	操作数 2（op2）	差值	标志位				说明	
				CF	OF	SF	ZF	有符号数	无符号数
1	3B	3B	00	0	0	0	1	op1 = op2	op1 = op2
2	3B	15	26	0	0	0	0	op1 > op2	op1 > op2
3	15	3B	DA	1	0	1	0	op1 < op2	op1 < op2
4	F9	F6	03	0	0	0	0	op1 > op2	op1 > op2
5	F6	F9	FD	1	0	1	0	op1 < op2	op1 < op2
6	15	F6	1F	1	0	0	0	op1 > op2	op1 < op2
7	F6	15	E1	0	0	1	0	op1 < op2	op1 > op2
8	68	A5	C3	1	1	1	0	op1 > op2	op1 < op2
9	A5	68	3D	0	1	0	0	op1 < op2	op1 > op2

标志位的值表示了一种什么关系呢？是相等？小于？还是大于？相等的情况比较简单：不管是有符号数还是无符号数，当且仅当操作数 1 和操作数 2 的值相等时，ZF 标志位置 1。表中的例 1 就是这种情况，小于和大于的情况则需要进一步的分析。

首先考虑小于的情况。当操作数 1 小于操作数 2 时，看起来似乎有借位，应该将进位标志位置 1。如果操作数是无符号数，这样做逻辑上是正确的。表中例 3、5、6 和 8 都是把操作数作为无符号数，且 op1 < op2，这样的情况下，确实是 CF=1。因此，对于无符号数，CF = 0 意味着 op1 ≥ op2。对于无符号数，严格意义上的不相等是指 CF=0 且 ZF=0，也就是 op1 ≥ op2 并且 op1 ≠ op2 。

例 3、5、7 和 9 是把操作数 op1 和 op2 当成有符号数，且 op1 < op2，这时 SF ≠ OF。在剩余例子中，SF = OF，而且操作数 op1 和 op2 是有符号数，op1 ≥ op2。对于无符号数，严格意义上的不相等是指 SF = OF，而且 ZF=0。也就是 op1 ≥ op2，并且 op1 ≠ op2 。

图 4-6 列出了 cmp 指令。回顾一下图 3-7，图 4-6 各个列的内容看起来几乎与 sub 指令是完全一样的。对于某些操作数的组合还有一些可选的操作码，这里列出的都是 MASM 6.11 使用的操作码。

一些要说明的问题是立即操作数。这些可以根据个人爱好来选择作为数的基数或字符来编码。假定 pattern 在数据段中指向一个字，那么，下面任何一种方式都是允许的。

操作数 1	操作数 2	字节数	操作码
8 位寄存器	8 位立即数	3	80
16 位寄存器	8 位立即数	3	83
32 位寄存器	8 位立即数	3	83
16 位寄存器	16 位立即数	4	81
32 位寄存器	32 位立即数	6	81
AL	8 位立即数	2	3C
AX	16 位立即数	3	3D
EAX	32 位立即数	5	3D
存储器字节	8 位立即数	3+	80
存储器字	8 位立即数	3+	83
存储器双字	8 位立即数	3+	83
存储器字	16 位立即数	4+	81
存储器双字	32 位立即数	6+	81
8 位寄存器	8 位寄存器	2	38
16 位寄存器	16 位寄存器	2	3B
32 位寄存器	32 位寄存器	2	3B
8 位寄存器	存储器字节	2+	3A
16 位寄存器	存储器字	2+	3B
32 位寄存器	存储器双字	2+	3B
存储器字节	8 位寄存器	2+	38
存储器字	16 位寄存器	2+	39
存储器双字	32 位寄存器	2+	39

图 4-6 cmp 指令

```
cmp             eax,    356
cmp             pattern,    od3a6h
cmp             bh,     '$'
```

注意，立即数必须是第二个操作数。指令

```
cmp     100,  total             ;  illegal
```

是不允许的，因为第一个操作数是立即数。

最后，图 4-7 列出了一些条件转移指令。这些指令中，许多指令有可供选择的助记符，但这些助记符能生成完全一样的机器代码；并且可用不同的方式描述同样的设置条件。在同一给定的设计中，通常使用一个助记符比使用不同助记符更加自然。

条件转移指令总是将第一个操作数与第二个操作数进行比较。例如，对于指令 jg，它就是“大于则转移”的意思，是指如果操作数 1 > 操作数 2，则转移。

任何条件转移指令都不会改变标志位的值。每一条指令都有短和近两种转移形式。和短的

对无符号操作数比较后的用法				
			操作码	
助记符	描述	转移标志	短	近
ja	大于则转移	CF=0	77	0F 87
jnbe	不小于或等于则转移	and ZF = 0		
jae	大于或等于则转移	CF = 0	73	0F 83
jnb	不小于则转移			
jb	小于则转移	CF = 1	72	0F 82
jnae	不大于或等于则转移			
jbe	小于或等于则转移	CF = 1	76	0F 86
jna	不大于则转移	or ZF = 1		
对有符号操作数比较后的用法				
			操作码	
助记符	描述	转移标志	短	近
jg	大于则转移	SF=OF	7F	0F 8F
jnle	不小于或等于则转移	and ZF=0		
jge	大于或等于则转移	SF=OF	7D	0F 8D
jnl	不小于则转移			
jl	小于则转移	SF ≠ OF	7C	0F 8C
jnge	不大于或等于则转移			
jle	小于或等于则转移	SF ≠ OF	7E	0F 8E
jng	不大于则转移	or ZF=1		
其他条件转移				
			操作码	
助记符	描述	转移标志	短	近
je	等于则转移	ZF = 1	74	0F 84
jz	为 0 则转移			
jne	不等于则转移	ZF = 0	75	0F 85
jnz	不为 0 则转移			
js	符号位为 1 则转移	SF=1	78	0F 88
jns	符号位不为 1 则转移	SF=0	79	0F 89
jc	有进位则转移	CF=1	72	0F 82
jnc	无进位则转移	CF=0	73	0F 83
jp	为偶数则转移	PF=1	7A	0F 8A
jpe	为偶数则转移			
jnp	不为偶数则转移	PF=0	7B	0F8B
jpo	为奇数则转移			
jo	溢出则转移	OF=1	70	0F 80
jno	无溢出则转移	OF=0	71	0F 81

图 4-7 条件转移指令

无条件转移指令一样，一条短的条件转移可使用单个字节的偏移量，可以控制转移到指令本身后面的 127 字节的地址，或之前的 128 字节的地址。一个短的条件转移指令需要两个字节的目

标代码：一个用于操作码，一个用于偏移量。一个近的条件转移指令可使用32位的偏移量，以及2个字节的操作码，因此，其总长度为6个字节。它能把控制转移到向后的2 147 483 648字节的地址，或向前的2 147 483 647字节的地址。

再举一些例子来说明有符号数和无符号数比较之后条件转移指令用法的区别。假设在EAX中存储了一个值，当这个值大于100时，需要采取一些措施。如果这个值是无符号数，那么可以用如下代码：

```
cmp             eax,   100
ja              bigger
```

对于任何大于00000064_{16}的值，其中包括在80000000_{16}和负的二进制补码数$FFFFFFFF_{16}$之间的值，都可执行转移。如果在EAX中的值是有符号数，那么指令

```
cmp            eax,  100
jg             bigger
```

是合适的。只有当值在00000065和7FFFFFFF之间时，而不是值为负的二进制补码时，转移才会发生。

现在来看看实现if结构的三个例子。该实现与高级语言编译器所采用的方式是一致的。首先考虑设计

```
if value < 10
then
      add 1 to smallCount;
else
      add 1 to largeCount;
end if;
```

假设value存储在EBX中，并且smallCount和largeCount指向存储器中的字。下面的80x86代码能实现这个设计：

```
            cmp  ebx, 10          ; value < 10 ?
            jnl  elseLarge
            inc  smallCount       ; add 1 to smallCount
            jmp  endValueCheck
elseLarge:  inc  largeCount       ; add 1 to largeCount
endValueCheck:
```

注意，此代码是完全独立的（自包含的），至于这部分设计之前或之后的整个设计是什么，不需要了解。可是，为了避免重复和保留字，要注意标号的使用。编译器通常会在一序列数字后生成一个包含字母的标号，但是，大多数情况下，程序员编码能做得更好。

现在考虑设计

```
if (total ≥ 100) or (count = 10)
then
      add value to total;
end if;
```

假设total和value是存储器中的双字，count存储在ECX寄存器中。实现该设计的汇编语言代码如下：

```
            cmp  total, 100       ; total >= 100 ?
            jge  addValue
            cmp  ecx, 10          ; count = 10 ?
            jne  endAddCheck
addValue:   mov  ebx, value       ; copy value
            add  total, ebx       ; add value to total
endAddCheck:
```

注意，这个设计中的or条件选择需要两个cmp指令。如果其中任一条件满足，都能执行相加指令。（为什么要用两条语句实现相加过程？为什么不用add total,value?）这个代码实现了一个条件选择捷径——如果第一个条件成立，那么第二个条件根本就不需要检查。如果用某些语言来设计代码，即使第一个条件成立，代码还是对一个or操作的两个操作数进行检查。

最后，考虑设计

```
if (count > 0) and (ch = backspace)
then
    subtract 1 from count;
end if;
```

对于第三个例子，假设count是在ECX寄存器中，并且ch是在AL寄存器中。这个设计可以采用如下方式实现：

```
            cmp  cx, 0            ; count > 0 ?
            jng  endCheckCh
            cmp  al, 08h          ; ch a backspace?
            jne  endCheckCh
            dec  count            ; subtract 1 from count
endCheckCh:
```

这个复合条件用了and，因此两个条件必须同时成立才能执行这段程序。这段代码用了一条and捷径——如果第一个条件不成立，那么根本不会检查第二个条件。如果用某些语言来设计代码，即使第一个条件不成立，代码还是对一个and操作的两个操作数都进行检查。

练习 4.2

1. 假定下面每小题，EAX寄存器包含00 00 00 4F，value指向的双字是FF FF FF 38 。确定每个条件转移语句是否转移到dest。

```
a. cmp   eax, value      b. cmp   eax, value
   jl    dest                jb    dest
c. cmp   eax, 04fh       d. cmp   eax, 79
   je    dest                jne   dest
e. cmp   value, 0        f. cmp   value, -200
   jbe   dest                jge   dest
g. add   eax, 200        h. add   value, 200
   js    dest                jz    dest
```

2. 下面的每一小题，都用了 if 结构，并且给出了汇编语言程序中变量的存储方式。写出一段汇编语言代码实现这个设计。

a. 设计：

```
if count = 0
then
    count := value;
end if;
```

假设：count 在 ECX 中；value 是存储器中的双字。

b. 设计：

```
if count > value
then
    count := 0;
end if;
```

假设：count 在 ECX 中；value 是存储器中的双字。

c. 设计：

```
if a + b = c
then
    check := 'Y';
else
    check := 'N';
end if;
```

假设：其中 a、b 和 c 都是存储器中的双字；字符 check 存放在 AL 寄存器中。

d. 设计：

```
if (value≤-1000) or (value≥1000)
then
    value := 0;
end if;
```

假设：value 在 EDX 中。

e. 设计：

```
if (ch≥'a') and (ch≤'z')
then
    add 1 to lowerCount;
else
    if (ch≥'A') and (ch≤'Z')
    then
        add 1 to upperCount;
    else
        add 1 to otherCount;
    end if;
end if;
```

假设：ch 在 AL 中；每个 lowerCount、upperCount 和 otherCount 都是存储器中的双字。

4.3 循环结构的实现

大多数程序都包含循环结构，常用的循环结构包括 while、until 和 for 循环。本节讨论如何

用 80x86 汇编语言实现这三种结构。下一节讨论其他一些实现 for 循环的指令。

一个 while 循环可以通过下面的伪代码设计来实现：

while *continuation condition* loop
... { body of loop }
end while;

首先检查循环继续条件，即布尔（Boolean）表达式，如果布尔表达式的值为真，那么循环体将被执行。然后再检查循环继续条件。当布尔表达式的值为假时，将继续执行循环体外（end while 后面）的语句。

用 80x86 实现 while 循环，大多采用以下形式：

```
while:      .           ; code to check Boolean expression
            .
            .
body:       .           ; loop body
            .
            .
          jmp  while    ; go check condition again
endWhile:
```

通常要用多个语句来检查布尔表达式的值。如果确定布尔表达式的值为假，则转移到 endWhile。否则，要么执行循环体，要么转移到它的标号处。注意：循环体的最后是一个 jmp 语句，以便再次检查循环继续条件。有两个常见的错误，要么忽略这个转移，要么转移到循环体。

由于 while 是 MASM 中的保留字，因此，在实际编写代码时，标号 while 并不允许使用。事实上，MASM 6.11 有一条 WHILE 指示性语句，它简化了 while 循环代码。但本书没有用到这条语句，因为本书所关心的是在机器语言级如何实现循环结构。

例如，用 80x86 汇编语言对以下设计进行编码。

while (sum < 1000) loop
... { body of loop }
end while;

假设 sum 是存储器中的双字，一种可能的实现方式是：

```
whileSum:   cmp   sum, 1000       ; sum < 1000?
            jnl   endWhileSum     ; exit loop if not
             .                    ; body of loop
             .
             .
            jmp   whileSum        ; go check condition again
endWhileSum:
```

语句

```
jnl     endWhileSum
```

可以直接实现整个设计，另一种可选的方式：

```
jge   endWhileSum
```

如果 sum ≥ 1000，则将控制转移到循环的最后。这是由于 sum < 1000 不成立时，恰好不等式 sum ≥ 1000 成立。但是 jnl 助记符不用颠倒不等式，就可以很容易地实现这个结构。

用一个例子来显示一个完整的循环体。假设要找出一个整数 x，它是一个以 2 为底的正整数 *number* 的对数，其最大整数 x 使得 $2^x \leq number$ 成立。实现该设计的程序如下：

```
x := 0;
twoToX := 1;
while twoToX≤number loop
      multiply twoToX by 2;
      add 1 to x;
end while;
subtract 1 from x;
```

假设 number 是存储器中的双字，下列 80x86 代码实现这个设计。twoToX 用 EAX 寄存器，x 用 ECX 寄存器。

```
            mov   ecx, 0       ; x := 0
            mov   eax, 1       ; twoToX := 1
whileLE:    cmp   eax, number  ; twoToX <= number?
            jnle  endWhileLE   ; exit if not
body:       add   eax, eax     ; multiply twoToX by 2
            inc   ecx          ; add 1 to x
            jmp   whileLE      ; go check condition again
endWhileLE:
            dec   ecx          ; subtract 1 from x
```

图 4-8 给出了运行这段代码的例子。存储在存储器中的 number 值为 750。在 INVOKE 指令处设置了一个断点，程序可以从开始处执行到该断点。ECX 寄存器中的值是 9，因为 $2^9 = 512$，2^9 小于 750。因此，ECX 寄存器中的 9 是一个正确的值。

通常 while 的循环条件是复合的，用布尔运算符 and 或者 or 连接两部分。对于 and 运算，其运算符的两边必须同时为真，循环条件才成立。对于 or 运算，只有两边的运算同时为假，循环条件才不成立。

修改前面的例子，使其包含一个复合条件。假设对下列设计编码：

```
while (sum < 1000) and (count≤24) loop
      ... { body of loop }
end while;
```

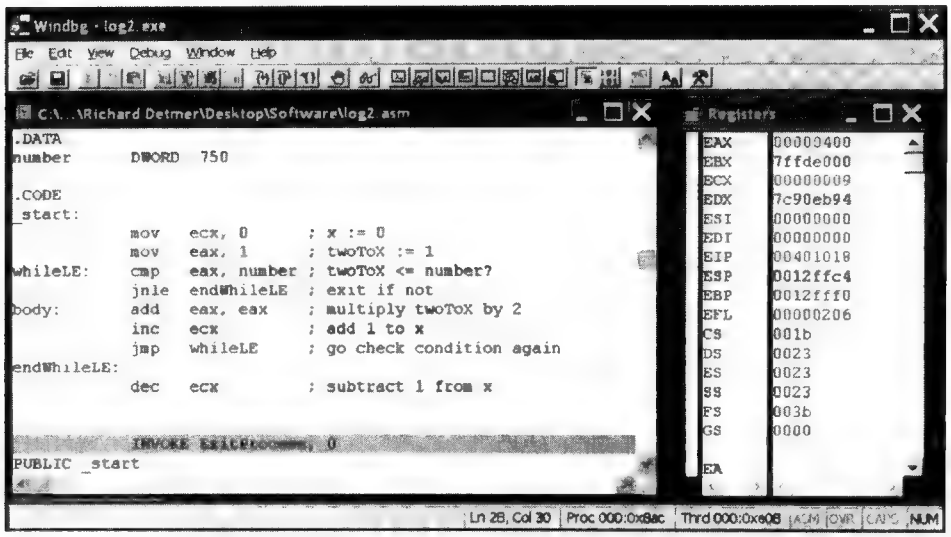


图 4-8 计算 $\log_2(number)$

假定 sum 是寄存器中的双字，count 值在 ECX 中，一种实现是：

```
whileSum:   cmp   sum, 1000        ; sum < 1000?
            jnl   endWhileSum      ; exit if not
            cmp   ecx, 24          ; count <= 24
            jnle  endWhileSum      ; exit if not
             .                     ; body of loop
             .
             .
            jmp   whileSum         ; go check condition again
endWhileSum:
```

再次修改例子，这次是用 or 而不是用 and：

```
while (sum < 1000) or (flag = 1) loop
    ... { body of loop }
end while;
```

现在，假设 sum 在 EAX 寄存器中，flag 是 DH 寄存器中的单字节。下面是用 80x86 实现整个设计的代码：

```
whileSum:   cmp   eax, 1000        ; sum < 1000?
            jl    body             ; execute body if so
            cmp   dh,1             ; flag = 1?
            jne   endWhileSum      ; exit if not
body:        .                     ; body of loop
             .
             .
            jmp   whileSum         ; go check condition again
endWhileSum:
```

注意上面两个例子的区别，对于 and 循环，只要复合条件中的任一运算不成立，即退出循环。对于 or 循环，只要复合条件中有一个运算成立，即执行循环体。

for 循环是一个计数器控制循环，在给定范围内，每执行一次循环，计数一次。在一些高级语言中，循环计数不仅可以是整数，还可以是其他类型。在汇编语言中，计数通常是整数。一个 for 循环以用下面的伪代码描述。

```
for index := initialValue to finalValue loop
    ... { body of loop }
end for;
```

一个 for 循环很容易就能转换到 while 结构。

```
index := initialValue;
while index≤finalValue loop
    ... { body of loop }
    add 1 to index;
end while;
```

这样，while 结构就能用 80x86 汇编语言来编码了。

until 循环是一种后检查条件的循环，即在执行循环体之后，再检查条件。通常，一个 until 循环可以通过下面的伪代码来表示：

```
until termination condition loop
    ... { body of loop }
end until;
```

循环体至少执行一次，然后检查结束条件。如果为假，则再次执行循环体；如果为真，则继续执行 end until 后面的语句。也可以采用下面的伪代码：

```
repeat
    ... { body of loop }
until termination condition;
```

它是在循环体执行后，再检查布尔表达式。

一个 until 循环的 80x86 实现，通常代码段如下：

```
until:      .           ; start of loop body
            .           .
            .           .
            .           ; code to check termination condition
endUntil:
```

如果检查结束条件，判断其条件为假，那么就会转移到 until。如果判断其条件为真，则要么转移到 endUntil，要么转移到 endUntil 标号处。

其他循环结构也能用汇编语言编码，forever 循环是最常用的。如果它出现在伪代码中，它总有一条 exit loop（退出循环）语句，以转移控制到循环结束，这通常是有条件的，在循环中要用到 if 语句。

本节最后举一个完整的例子。假定有一份工作协议，同意第一天支付雇员 1 分工资，第二天支付 2 分，第三天支付 4 分，第四天支付 8 分，以此类推，雇员的薪水每天都会翻倍增长。按照这个合同，必须工作多少天雇员可以成为百万富翁？

显然，在这个方案的设计中需要使用循环。采用计数器控制的循环结构是不合适的，因为我们不知道要重复多少次。通过合理的结构化设计，决定使用 while 循环或 until 循环。下面是个使用 while 循环的伪代码：

```
nextDaysWage := 1;
totalEarnings := 0;
day :=0;
while totalEarnings < 100000000 loop
    add nextDaysWage to totalEarnings;
    multiply nextDaysWage by 2;
    add 1 to day;
end loop;
```

注意，最后收入的总和要计算到分；100 万美元可以转换成 1 000 000 000 分。要实现这个设计，把 totalEarnings 存在寄存器 EAX 中，nextDaysWage 存在 EBX 中，天数 day 存在 ECX 中。图 4-9 是程序执行到退出点时，Windbg 调试器的屏幕截图。赚 100 万美元需要多少天？最后一天的工资是多少？这些天的总收入是多少？

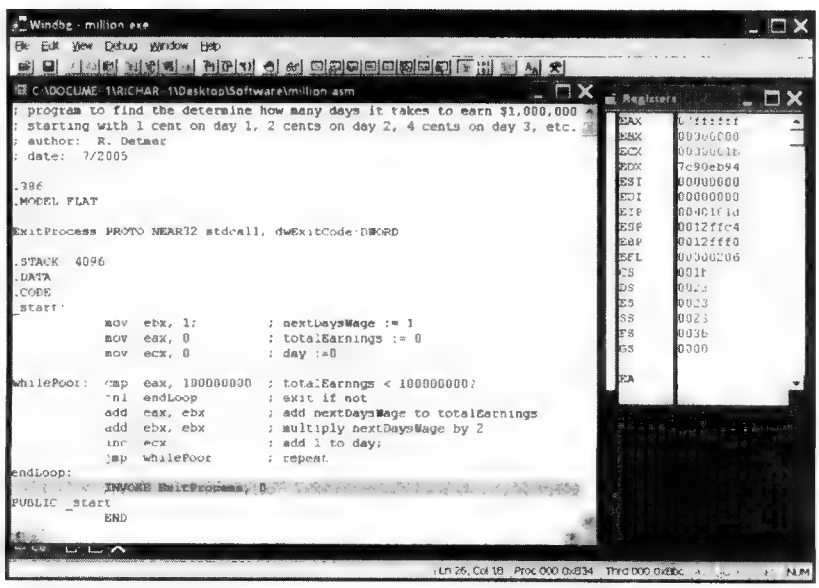


图 4-9 成为百万富翁的例子

练习 4.3

1. 下面每一小题都含一个 while 循环。假设 sum 是数据段中的双字，count 的值在 ECX 寄存器中。给出实现以下设计的相应的 80x86 代码。

a.
```
sum := 0;
count := 1;
while (sum < 1000) loop
    add count to sum;
    add 1 to count;
end while;
```

b.
```
sum := 0;
count := 1;
while (sum < 1000) and (count ≤ 50) loop
    add count to sum;
    add 1 to count;
end while;
```

c.
```
sum := 0;
count := 100;
while (sum < 1000) or (count ≥ 0) loop
    add count to sum;
    subtract 1 from count;
end while;
```

2. 下面每一小题都含一个 until 循环。假设 sum 是数据段中的双字，count 的值在 ECX 寄存器中。给出实现以下设计的相应的 80x86 代码。

a.
```
sum := 0;
count := 1;
until (sum > 5000) loop
    add count to sum;
    add 1 to count;
end until;
```

b.
```
sum := 0;
count := 1;
until (sum > 5000) or (count = 40) loop
    add count to sum;
    add 1 to count;
end until;
```

c.
```
sum := 0;
count := 1;
until (sum ≥ 5000) and (count > 40) loop
    add count to sum;
    add 1 to count;
end until;
```

3. 下面每一小题都含一个 for 循环。假设 sum 是数据段中的双字，count 的值在 ECX 寄存器中。给出实现以下设计的相应的 80x86 代码。

```
a.  sum := 0;
    for count := 1 to 100 loop
        add count to sum;
    end for;
b.  sum := 0;
    for count := -10 to 50 loop
        add count to sum;
    end for;
c.  sum := 1000;
    for count := 100 downto 50 loop
        subtract 2*count from sum;
    end for;
```

编程练习 4.3

1. 修改图4-9程序的设计，用until循环代替while循环。用80x86汇编语言程序实现修改后的设计，并运行它。
2. 修改图 4-9 程序的设计，问赚 400 万美元所需要的天数。提示：如果只是简单地更改 cmp 指令中的立即数的值，那么程序将会出错。必须对程序做其他的更改，程序才能正常运行。在这个设计中，能赚的最大数目是多少？
3. 用 80x86 汇编语言程序设计和实现，求从 1 到 1000 的和。
4. 用 80x86 汇编语言程序设计和实现，求 $1+2+\cdots+n$，直到和为 1000 时最小的 n。
5. 用 80x86 汇编语言程序设计和实现，求 $1^2+2^2+3^2+\cdots+1000^2$ 的和。
6. 两个非负整数的最大公约数是这两个数的最大除数。下面的算法能找出 number1 和 number2 的最大公约数。

```
gcd := number1;
remainder := number2;

until (remainder = 0) loop
    dividend := gcd;
    gcd := remainder;
    remainder := dividend mod gcd;
end until;
```

写一个能实现这个设计的 80x86 汇编语言程序，假设 number1 和 number2 是数据段中的双字。用几组不同的数据测试你的程序。

4.4 汇编语言的 for 循环

通常，如果要执行的循环体的次数是已知的，那么可以在写程序时，用常量来表示循环次数；或者在执行循环之前，用已赋值的变量值表示循环次数。对编写这样的循环，for 循环结构是最理想的。

上一节提到如何将一个 for 循环转换为一个 while 循环。这个方法很有用，并且也是实现 for 循环最常用的方法。但是，80x86 微处理器还有一些指令，它们可使某些 for 循环编码变得更简单。

考虑下面的两个 for 循环例子，第一个例子是向上计数，第二个例子是向下计数。

第一个 for 循环：

```
for index := 1 to count loop
    ... { body of loop }
end for;
```

第二个 for 循环：

```
for index := count downto 1 loop
    ... { body of loop }
end for;
```

每个循环体执行的次数是 count。如果 index 的值不需要在循环体内显示或计算，那么向下计数的循环和向上计数的循环是一样的，尽管向上计数的循环的设计有些不自然，向下计数的 for 循环很容易用 80x86 汇编语言中的 loop 指令实现。

loop 指令有如下的格式：

```
loop    statementLabel
```

其中 statementLabel 是语句的标号，是 loop 指令的较短偏移量(向后 128 字节或向前 127 字节)。loop 指令占用两个字节的目标代码，操作码 E2 和一个字节的偏移量。

loop 指令会引起下面的操作：

- ECX 中的值是递减的。
- 如果 ECX 中新的值是 0，那么继续执行 loop 指令下面的语句。
- 如果 ECX 中新的值不是 0，那么执行转移指令到 statementLabel。

虽然 ECX 寄存器是一个通用寄存器，但是在 loop 指令和其他一些指令中，它可以作为计数器因而有特殊地位。在这些指令中，没有任何寄存器可以替代 ECX。实际上，这意味着编写循环代码时，ECX 不能用作其他用途。

向下计数的 for 循环结构：

```
for count := 20 downto 1 loop
    ... { body of loop }
end for;
```

使用 loop 指令，80x86 汇编语言所编写的代码如下：

```
            mov   ecx, 20        ; number of iterations
forCount:    .                   ; body of loop
             .
             .
            loop  forCount       ; repeat body 20 times
```

第一次执行循环体时，ECX 寄存器中的计数值是 20， loop 指令执行后，计数值减到 19。19 不等于 0，因此，控制转移到标号为 forCount 的循环体开始处。第二次执行循环体时，ECX 寄存器中的值还是 19。最后一次执行循环体时，ECX 中的值是 1。loop 指令执行之后，ECX 的值将为 0，循环不再转移到 forCount 处。loop 后面的指令将继续执行。

显然，标志 for 循环体的记号用 for 合适，但是，它是 MASM 中的保留字，用来简化 for 循环编码的指令。再强调一次，本文重点是了解计算机如何在机器层工作，因此，不必用这条指令。

现在，假设 number 指向的存储器中的双字是循环的执行次数，实现向下计数的 for 循环的 80x86 代码如下：

```
            mov   ecx, number    ; number of iterations
forIndex:    .                   ; body of loop
             .
             .
            loop  forIndex       ; repeat body number times
```

只有当 number 中的值不为 0 时，这段代码才是可靠的。如果 number 中的值为 0，那么执行循环体后，0 被减为 FFFFFFFF（做这个减法需要借位），再执行一次循环体，FFFFFFFF 减为 FFFFFFFE，以此类推。在 ECX 中的值回到 0 之前，循环体要执行 4 294 967 296 次！为了避免这样的问题，代码可以写为：

```
            mov   ecx, number    ; number of iterations
            cmp   ecx, 0         ; number = 0 ?
            jz    endFor         ; skip loop if number = 0
forIndex:    .                   ; body of loop
             .
             .
            loop  forIndex       ; repeat body number times
endFor:
```

如果 number 中的数是一个有符号数，并且是负的，那么

```
jle   endFor        ; skip loop if number <= 0
```

是一个更合适的条件转移。

还有一种方法保证 for 循环可靠，那就是当 ECX 中的值为 0 时，循环不会执行。80x86 体系结构，设置了一条 jecxz 条件转移指令，如果 ECX 寄存器中的值为 0，则转移到它的目的地。使用 jecxz 指令时，上面的例子可以这样编码：

```
            mov   ecx, number    ; number of iterations
            jecxz endFor         ; skip loop if number = 0
forIndex:    .                   ; body of loop
             .
             .
            loop  forIndex       ; repeat body number times
endFor:
```

jecxz 指令是双字节长，操作码 E3，加上一个字节的偏移量。和其他的条件转移指令一样，jecxz 不会影响标志位的值。

当循环体大于 127 字节时，jecxz 指令可用于实现向下计数的 for 循环。但对于 loop 指令的一个字节偏移量来说，大于 127 字节的循环体长度是太长了。例如，结构

```
for counter := 50 downto 1 loop
    ... { body of loop }
end for;
```

能够这样编码

```
            mov   ecx, 50        ; number of iterations
forCounter:  .                   ; body of loop
             .
             .
            dec   ecx            ; decrement loop counter
            jecxz endFor         ; exit if counter = 0
            jmp   forCounter     ; otherwise repeat body
endFor:
```

但是，由于 dec 指令将零标志位 ZF 置 1 或清 0，所以可以使用相对高效的条件转移指令 jz endFor 来代替 jecxz 指令。

即使 for 循环计数值增加，且必须在循环体内使用，用一个 loop 语句实来现 for 循环还是很方便的。当要用一个单独的计数器用作循环计数时，loop 语句用 ECX 控制循环次数。例如，以下 for 循环的实现

```
for index := 1 to 50 loop
    ...{ loop body using index }
end for;
```

ECX 寄存器可用来存储从 1 到 50 的 index 计数，同时 ECX 寄存器从 50 向下计数，直到 1。

```
        mov   ebx, 1      ; index := 1
        mov   ecx, 50     ; number of iterations for loop
forNbr:  .
         .                ; use value in EBX for index
         .
        inc   ebx         ; add 1 to index
        loop  forNbr      ; repeat
```

练习 4.4

1. 下面的每一小题都是用 loop 循环语句实现 for 循环。请问每个循环体执行多少次？

a.

```
    mov  ecx, 10
forA:  .
       .              ; body of loop
       .
      loop forA
```

b.

```
      mov  ecx, 1
forB:  .
       .              ; body of loop
       .
      loop forB
```

c.

```
      mov  ecx, 0
forC:  .
       .              ; body of loop
       .
      loop forC
```

d.

```
      mov  ecx, -1
forD:  .
       .              ; body of loop
       .
      loop forD
```

2. 下面每小题都有一个 for 循环。假设 sum 是数据段中的双字，给出能实现设计的 80x86 代码段，在代码中恰当地使用 loop 语句。

a.
```
sum := 0;
for count := 50 downto 1 loop
    add count to sum;
end for;
```

b.
```
sum := 0;
for count := 1 to 50 loop
    add count to sum;
end for;
```

c.
```
sum := 0;
for count := 1 to 50 loop
    add (2*count - 1) to sum;
end for;
```

编程练习 4.4

1. 假定 lastNbr 是存储器中的双字。设计和实现一个 80x86 的汇编语言程序，计算 $1^2+2^2+3^2+\cdots+\text{lastNbr}^2$ 的和。
2. 二项式系数$\binom{n}{k}$定义为$\binom{n}{k}=\frac{n!}{k!(n-k)!}$ $(0 \leqslant k \leqslant n)$。假定 n 和 k 的值是存储在存储器中的双字，

设计和实现一个计算$\binom{n}{k}$的 80x86 的汇编语言程序。提示：不要分别计算 n！和 k！，而是把$\frac{n!}{k!}$作为 $n\times(n-1)\times\cdots\times(k-1)$ 来计算。

4.5 数组

程序常用数组存储数据值的集合，通常用循环控制数组中的数据。通过程序数据段中的 DUP 指令可为数组预留存储空间。这一节讨论两种用 80x86 汇编语言访问一维数组的方法。

假定 nbrElts 双字整型数据集存储在内存单元 nbrArray 处，并且 nbrElts 的值也是存储器的双字。如果处理这个数组，首先求出这些数组元素的平均值，然后给小于平均值的元素加 10。下面的设计可实现整个功能：

```
{ find sum and average }
sum := 0;
get address of first item of array;
for count := nbrElts downto 1 loop
      add doubleword at address in array to sum;
      get address of next item of array;
end for;
average := sum/nbrElts;

{ add 10 to each array element below average }
get address of first item of array;
for count := nbrElts downto 1 loop
      if doubleword of array < average
      then
            add 10 to doubleword;
      end if;
      get address of next item of array;
end for;
```

该设计有一些奇怪的指令："取出数组中第一项的地址"和"取出数组中下一项的地址"，这些指令反映了特殊的汇编语言实现方式，如果现有的任务要求顺序移动数组中的数，那么用这种方式能很好地完成任务。能够这样做是因为 80x86 的寄存器间接寻址，有关寄存器间接寻址在第 2 章已经讨论过了。上例使用 EBX 寄存器保存当前存取的字的地址。回想一下，[ebx] 指 EBX 寄存器中存储的地址的双字，而不是 EBX 寄存器本身存储的双字。在 80x86 体系结构中，任何一个通用寄存器 EAX、EBX、ECX、EDX 以及索引寄存器 EDI 和 ESI 都可作为指针来使用。ESI 和 EDI 寄存器常为串所用，串通常是字符数组，本书没有讨论字符串操作。该程序如图 4-10 所示。

```
; given an array of doubleword integers, (1) find their average and
; (2) add 10 to each number smaller than average
; author:  R. Detmer
; date:  7/2005

.386
.MODEL FLAT

ExitProcess PROTO NEAR32 stdcall, dwExitCode:DWORD

.STACK      4096

.DATA
nbrArray    DWORD    25, 47, 15, 50, 32, 95 DUP (?)
nbrElts     DWORD    5

.CODE
_start:
; find sum and average
            mov     eax,0              ; sum := 0
            lea     ebx,nbrArray       ; get address of nbrArray
            mov     ecx,nbrElts        ; count := nbrElts
            jecxz   quit               ; quit if no numbers
forCount1:  add     eax,[ebx]          ; add number to sum
            add     ebx,4              ; get address of next item of array
            loop    forCount1          ; repeat nbrElts times

            cdq                        ; extend sum to quadword
            idiv    nbrElts            ; calculate average

; add 10 to each array element below average
            lea     ebx,nbrArray       ; get address of nbrArray
            mov     ecx,nbrElts        ; count := nbrElts
forCount2:  cmp     [ebx],eax          ; number < average ?
            jnl     endIfSmall         ; continue if not less
            add     [ebx], 10          ; add 10 to number
endIfSmall:
            add     ebx,4              ; get address of next item of array
            loop    forCount2          ; repeat

quit:       INVOKE ExitProcess, 0      ; exit with return code 0

PUBLIC _start                          ; make entry point public
            END                        ; end of source code
```

图 4-10　处理数组的例子

为了实现程序，在 nbrArray 中允许 100 个双字的空间，但是只使用前面 5 个。计算的和存储在寄存器 EAX 中，执行除法后，平均值也放在同一个寄存器。图 4-11 显示了在 Windbg 中程序的执行，第一个循环开始之前，寄存器的内容如图中右边的界面。注意 ECX 存放的是数据元素的数目，EBX 中存放的是 00404000，和存储器所显示的 nbrArray（&nbrarray）地址是同一个地址。lea 指令把这个地址放在 EBX 中。

图 4-11 建立数组处理的循环

助记符 lea 表示“取有效的地址”。lea 指令的格式如下：

```
lea  destination, source
```

目的操作数通常是一个 32 位的通用寄存器，源操作数是存储器的值。源操作数的地址放在寄存器中(参照mov指令,mov destination,source,其中源操作数的地址的值复制到目的地址)。lea 指令的操作码是 8D。

注意，EBX 中的当前元素的地址加 4 后就是数组中的下一个元素的地址。重复循环，观察 EBX 的变化，下一个数组元素被加到 sum。然后，在第二个 lea 指令处设置断点，并且完成第一次循环。图 4-12 是 Windbg 窗口。EAX 中存放的是 21_{16}，这是正确的，因为数组元素的平均值是 33.8。

最后，程序的结束处设置一个断点。图 4-13 是最后显示的内容。注意，存储器中的 19_{16} 已经改为 23_{16}，$0f_{16}$ 已经变为 19_{16}，并且 20_{16} 已经改变为 $2a_{16}$，也就是说，每个小于平均值 21_{16} 的都加了 10。

如果用 C++ 之类的高级语言编写这个程序，第一个循环的设计应该是：

```
for index := 0 to nbrElts-1 loop
    add nbrArray[index] to sum;
end for;
```

图 4-12　数组元素的平均值

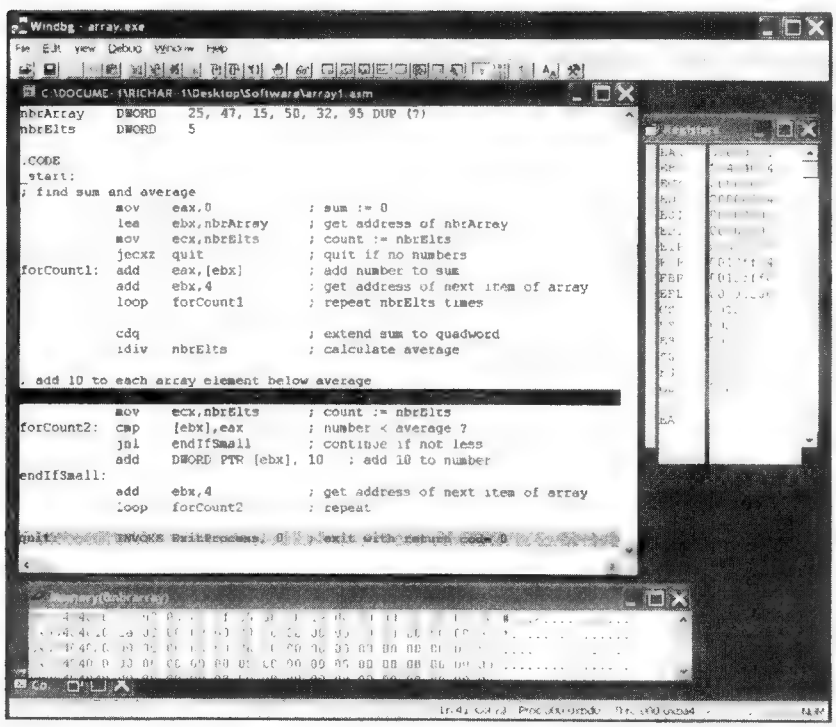


图 4-13　改变的数组元素最小值

图 4-14 是一段直接实现这个设计的 80x86 程序代码。这里只显示部分代码，因为剩余的部分与图 4-10 程序的第一个版本一样。不同的是，在上面的设计中，ECX 执行的是 index 的功能，也就是从 0 计数到 nbrElts−1。这里使用**索引寻址**而不是寄存器间接寻址来指向数组的元素。地址格式 nbrArray[4*ecx] 被编译成带有位移的地址，这个位移就是 nbrArray 的地址。ECX 当作一个索引寄存器，索引的**比例因子**是 4。程序执行时，所使用的操作数的地址是位移量与 4 倍索引寄存器内容的和。换句话说，第一个操作数在 nbrArray+0，第二个在 nbrArray+4，等等。使用这种索引寻址的优点在于不一定要按顺序来访问数组元素。

```
.CODE
_start:
; find sum and average
            mov     eax,0               ; sum := 0
            mov     ecx,0               ; index := 0
for1:       cmp     ecx,nbrElts         ; index < nbrElts
            jnl     endFor1             ; exit if not
            add     eax,nbrArray[4*ecx]   ; add number to sum
            inc     ecx                 ; increment index
            jmp     for1                ; repeat
endFor1:
            cdq                         ; extend sum to quadword
            idiv    nbrElts             ; calculate average

; add 10 to each array element below average
            mov     ecx,0               ; index := 0
for2:       cmp     ecx,nbrElts         ; index < nbrElts
            jnl     endFor2             ; exit if not
            cmp     nbrArray[4*ecx],eax   ; number < average ?
            jnl     endIfSmall          ; continue if not less
            add     DWORD PTR nbrArray[4*ecx], 10   ; add 10 to number
endIfSmall:
            inc     ecx                 ; increment index
            jmp     for1                ; repeat
endFor2:

quit:       INVOKE ExitProcess, 0       ; exit with return code 0
```

图 4-14　使用索引地址处理数组

80x86 体系结构还有一些其他的寻址模式，最复杂的是**基址加索引**组合方式。比如 100[ebx+8*ecx]，就是一种汇编语言寻址模式。其中，操作数地址的计算是 EBX 的内容加上 8 倍的 ECX 内容后加 100。注意这里，位移量 100 实际上与前面例子中开始地址 nbrArray 的作用是一样的。所允许的比例因子是 2、4 和 8。

练习 4.5

1. 修改图 4-10 的程序，把第二个循环改为大于平均值的数组元素置 0。
2. 修改图 4-10 的程序，把第二个循环改为与平均值相差在 5 的范围之内的数组元素。包括等于 average−5 和 average+5 的元素。

编程练习 4.5

1. 假定要处理一个数组，但是数组元素的个数并不确定，用 −9999 作为数据结束的标志值，但该值不是数组元素中的值。修改图 4-10 的程序，使之包含下面的代码段：

```
.DATA
nbrArray    DWORD    25, 47, 15, 50, 32, -9999, 94 DUP (?)
nbrElts     DWORD    ?
```

改变程序，这样第一个循环是 while 循环，它不仅计算和，而且还要计算数组中有多少个元素，并把计算的值存放到 nbrElts 中。

2. 许多高级语言用字节数组来存储字母字符串，使用空字节 null（00）来标记字符串的结束。假定存储在存储单元的 charStr 的字符串是以空字符 null 结束，该字符串包含的字母既有大写字母，也有小写字母。写一个 80x86 汇编语言程序，把 charStr 中的每个大写字母转换成小写字母，其余的字母保持不变。
3. 判断素数有多种方法。下面是一个找出前 100 个素数的设计。用 80x86 汇编语言实现这个设计。

```
prime[1] := 2; { first prime number }
prime[2] := 3; { second prime number }
primeCount := 2;
candidate := 4; { first candidate for a new prime }
while primeCount < 100 loop
    index := 1;
    while (index ≤ primeCount)
            and (prime[index] does not evenly divide candidate) loop
        add 1 to index;
    end while;
    if (index > primeCount)
    then {no existing prime evenly divides the candidate, so it is a new
            prime}
        add 1 to primeCount;
        prime[primeCount] := candidate;
    end if;
    add 1 to candidate;
end while;
```

4.6 本章小结

这一章讨论了能用来实现许多高级设计或语言功能的 80x86 指令，包括 if 语句以及各种不同的循环结构和数组。

jmp 指令无条件地将控制转移到目标语句，它有几种形式：短跳，是指转移到 jmp 指令之前的 128 字节以内的目标地址，或者转移到 jmp 指令之后的 127 字节以内的目标地址；还有近跳，是指转移到距离目标地址 32 位偏移量的目标地址。jmp 指令可用在不同循环结构之中，通常可将控制转移到循环的开始。jmp 指令还可用在 if-then-else 结构中，在 then 代码的最后，将控制转移到 endif。这样，就不会执行 else 代码了。jmp 语句相当于大多数高级语言中的 goto 语句。

条件转移语句检查标志寄存器中一个或多个标志位的设置，然后根据标志值判断是转移到目标语句，或继续执行下面的指令。条件转移指令按照偏移量，有短跳和近跳两种形式。条件转移指令有很多，在 if 语句和循环中，它常和比较指令一起检查布尔条件。

cmp（比较）指令的唯一目的就是将 EFLAGS 寄存器中的标志位置 1 或清 0。每条指令比较两个操作数，并设置标志位的值。通过从第一个操作数中减去第二个操作数来完成这两个数的比较。与 sub 指令的不同之处在于，相减的差值并不保留。比较指令通常用在条件转移指令之前。

像 while、until 和 for 这样的循环结构，能够用比较、转移和条件转移指令实现。loop 指令还提供了一种实现 for 循环的方法。为了使用 loop 指令，在循环开始之前，把一个计数器放在 ECX 寄存器中。loop 指令本身是在循环体的底部，它递减 ECX 中的值。如果新的值不是零，则将控制转移到目的地（通常是循环体的第一条语句），循环体执行的次数的初值放在 ECX 寄存器中。当计数器的初值为 0 时，条件转移指令 jecxz 可用来防止执行循环。

程序中数据段的 DUP 指令可为一个数组预留存储空间。数组中第一个元素的地址是在寄存器中，第一个元素的地址加上数组中元素的长度可以取出下一个元素。这样，就能顺序存取数组中的元素。当前的元素用寄存器间接寻址获得，通常用 lea（取有效地址）指令取出数组的初始地址。

第5章 过　　程

80x86 体系结构能够实现那些类似于高级语言的过程。本章主要讨论三个问题：如何从调用程序回到过程并返回；如何传递参数值给过程，并返回结果；如何编写独立于调用程序的过程代码。硬件堆栈的使用可以解决上述三个问题。本章首先讨论 80x86 堆栈（stack）。

5.1　80x86 堆栈

本书中的程序使用如下的代码来分配堆栈空间：

```
.STACK    4096
```

其中指示性指令 .STACK 告诉汇编器预留 4096 字节的未初始化存储空间。操作系统初始化堆栈指针寄存器 ESP 指向堆栈 4096 个字节之上的第一个字节地址。堆栈大小的分配取决于程序的需要。

堆栈常常用来压入或取出字或双字。压入和取出是作为执行调用及返回指令的一部分而自动完成（见 5.2 节）。也可以手动地执行 push（入栈）或 pop（出栈）指令。本节主要讨论 push 和 pop 指令结构，考察它们是如何影响堆栈的内容的。

push 指令的源代码语法如下：

```
push  source
```

其中源操作数（source）可以是一个 16 位寄存器、32 位寄存器、段寄存器，或者是存储器中的一个字、双字、一个字节的立即数、一个字的立即数或一个双字的立即数。只有一个字节的操作数是立即数，因此，多字节是以一个字节立即数的形式入栈的。图 5-1 列出了允许使用的操作数类型。push 指令的常用助记符就是 push，但是，如果操作数的大小不明确（可能是一个小的立即数），那么可以使用助记符 pushw 和 pushd 来分别指定字或双字的操作数。

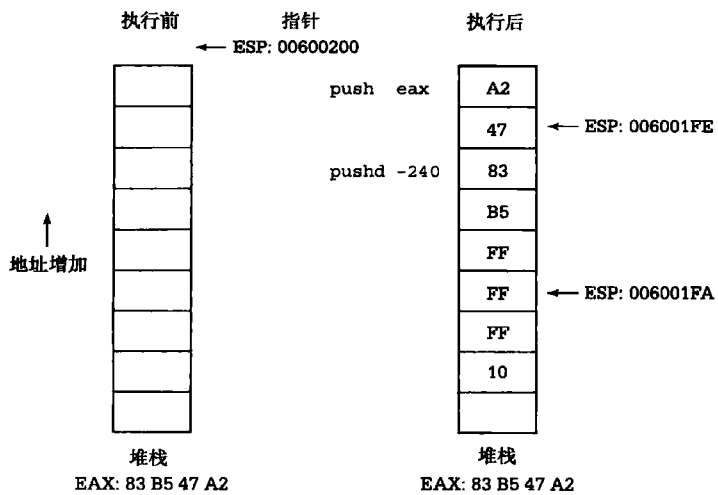
当一个双字操作数入栈时，堆栈指针 ESP 将减去 4。回想一下，最初 ESP 指向的是被分配空间之上一个字节的地址。减去 4 以后，ESP 将指向堆栈顶部的双字。然后，这个操作数被存储在 ESP 所指向的地址，也就是堆栈的顶部。如果是字操作数入栈，执行情况类似，但堆栈指针 ESP 是减去 2。值得注意的是，对于单字节的立即数存储，虽然指令中仅存放一个字节，但它扩展符号位为双字，实际上以双字的形式存储在堆栈内。一个字节的操作数节省了三个字节的目标代码，但在运行时，并没有节省堆栈空间。

操作数	字节数	操作码
寄存器		
EAX 或 AX	1	50
ECX 或 CX	1	51
EDX 或 DX	1	52
EBX 或 BX	1	53
ESP 或 SP	1	54
EBP 或 BP	1	55
ESI 或 SI	1	56
EDI 或 DI	1	57
段寄存器		
CS	1	0E
DS	1	1E
ES	1	06
SS	1	16
FS	2	0F A0
GS	2	0F A8
存储器字	2+	FF
存储器双字	2+	FF
字节立即数	2	6A
字立即数	3	68
双字立即数	5	68

图 5-1 push 指令

示例

下面是运行两个 push 指令的例子。假设，ESP 初始值是 00600200。第一个 push 指令将 ESP 减少到 06001FC 并把 EAX 内容存储到该地址。注意，在存储器中已预留低字节和高字节的空间。执行第二条 push 指令后，将 ESP 值减为 006001F8，并存储 FFFFFF10（-240_{10}）到该地址。



现在使用 Windbg 来跟踪这些指令的实际运行。从下面指令开始汇编后：

```
mov   eax, 83b547a2h
push  eax
pushd -240
```

汇编器显示：

```
00000000  B8 83B547A2  mov   eax, 83b547a2h
00000005  50           push  eax
00000006  68 FFFFFF10  pushd -240
```

这是从图 3-1 中 mov 和图 5-1 中 push 所对应的操作码得到的。Windbg 显示 EAX 寄存器初始值为 83b547a2（如图 5-2 所示）。因为关注的是栈顶的几个字节，所以需要将 ESP 值设为 0012ffc4（在其他时刻或其他电脑上可能是别的值）。为了显示栈顶的 16 个字节，需要在地址 0x0012ffb4 处打开一个存储器视图。这些字节显示在存储器窗口的顶行。注意，该堆栈包含“无用的”值。

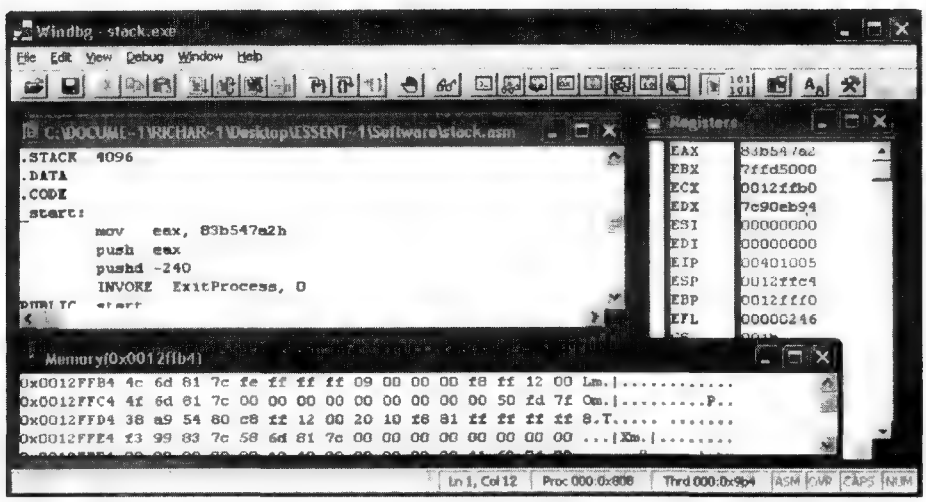


图 5-2　push 操作之前的堆栈测试

现在跟踪 push 指令的执行，结果如图 5-3 所示。注意：现在 ESP 值是 0012ffc0，即被减去 4。存储器上（突出显示）的最后四个字节表示存放在新栈指针地址上的双字。EAX 的字节按向后顺序保存到存储器。最后，考察一下 pushd 指令，执行的结果如图 5-4 所示。这时 ESP 的值为 0012ffbc，再减少 4。顶端内存线再次突出显示的值为 −240，同样 ffffff10 以相反方向存储。

随着操作数的入栈， ESP 的值将递减，并存储相应的新值。push 指令不会影响任何标志位。

注意，堆栈是“向下递减”的，这与常见的软件堆栈相反。同时，堆栈中最容易得到的值是最后一个被压入堆栈的，存放在 ESP 所指的地址处。另外，随着入栈和过程调用操作的执行，ESP 的值将频繁地改变。5.3 节将讨论如何利用 EBP 寄存器在堆栈中建立固定参考指针的方法，这样，在参考指针附近的值能够被读取，而不用弹出所要读取的值上面所有的值。

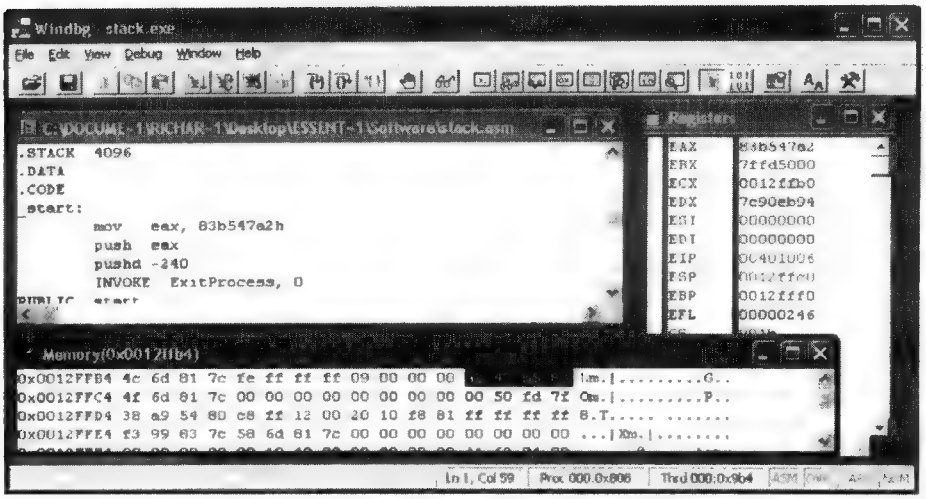


图 5-3 EAX 入栈

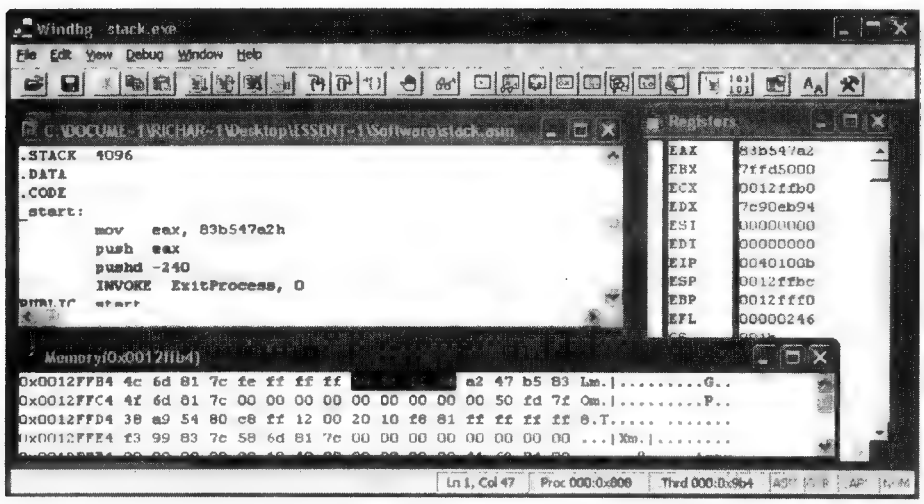


图 5-4 −240 入栈

pop 指令的功能与 push 指令相反。pop 指令的语法为：

```
pop  destination
```

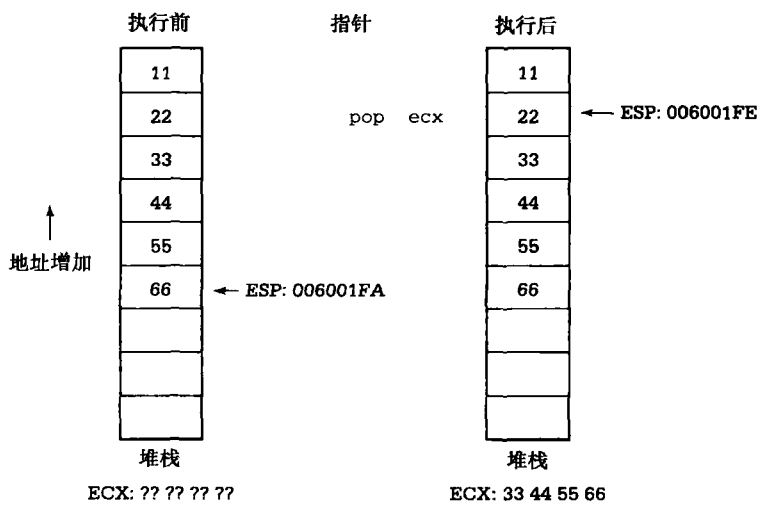
其中目的操作数（destination）指向存储器中的字或双字，或任何一个 16 位寄存器、32 位寄存器，或者是除 CS 之外的任何段寄存器（入栈指令可用 CS 寄存器）。对于双字出栈操作，pop 指令复制 ESP 所指的地址中的双字，并将它存放到目的地址中，然后 ESP 加 2。单字出栈情况类似，只是 ESP 的值是加 2。图 5-5 列出不同目的操作数的 pop 指令信息。

操作数	字节数	操作码
寄存器		
EAX 或 AX	1	58
ECX 或 CX	1	59
EDX 或 DX	1	5A
EBX 或 BX	1	5B
ESP 或 SP	1	5C
EBP 或 BP	1	5D
ESI 或 SI	1	5E
EDI 或 DI	1	5F
段寄存器		
DS	1	1F
ES	1	07
SS	1	17
FS	2	0F A1
GS	2	0F A9
存储器字	2+	8F
存储器双字	2+	8F

图 5-5　pop 指令

示例

这个例子说明了出栈指令是如何工作的。对于双字的出栈操作，先把 ESP 所指地址的内容复制到 ECX，然后 ESP 加 4。出栈的操作只是改变了数据在逻辑上的存放位置，并没有改变其在物理上的存放位置。注意：在 80x86 体系结构中，双字中的字节在存储器中是低字节优先。



push 和 pop 指令的一个用途是将寄存器中的内容暂时存放在堆栈中。前面已经提到，在编程时，寄存器是稀缺资源。例如，假定寄存器 EDX 已用来存储某些程序变量，但是在进行除法运算时，被除数必须先扩展存放在 EDX：EAX 中，为了避免丢失 EDX 中存储的数据值，必

须先将 EDX 中存放的数据值入栈。具体如下所示：

```
push  edx       ; save variable
cdq             ; extend dividend to quadword
idiv  Divisor   ; divide
pop   edx       ; restore variable
```

本例假设不需要将除法操作得到的余数存储到 EDX。如果确实需要存储，则在数据保存到 EDX 之前，可将余数复制到其他位置。

上例表明，push 和 pop 指令通常成对使用。在使用堆栈传递参数给过程时，如果堆栈中数据没有被复制到目的单元，那么这些数据就会丢失。

除了 push 和 pop 指令之外，还有一些特殊的助记符可用来压入和取出标志寄存器，例如 pushf（pushfd 用于扩展标志寄存器）和 popf（popfd 用于扩展标志寄存器）。图 5-6 对这些助记符进行归纳总结。通常这些助记符是在过程代码中使用。显然，popf 和 popfd 指令能够改变标志位的数值，它们是唯一能改变标志位的入栈或出栈指令。

在 80x86 体系结构中，有些可以使用单一的指令压入或取出所有通用寄存器的内容。pushad 指令能依次将数据压入 EAX、ECX、EDX、EBX、ESP、EBP、ESI 和 EDI。压入到 ESP 的值是所有寄存器被压入前的地址。popad 指令按照相反的顺序依次从同样的寄存器中提取数据，只不过 ESP 的值被丢弃。按相反顺序取出这些寄存器中的值，以保证 pushad 和 popad 是成对使用的，这样，每一个寄存器（除了 ESP 外）都能恢复它的初值（为什么？）。图 5-7 列出了所有的入栈和出栈指令，包括压入和取出 16 位寄存器的 pusha 和 popa 指令。

指令	字节数	操作码
pushf/pushfd	1	9C
popf/popfd	1	9D

图 5-6　pushf 和 popf 指令

指令	字节数	操作码
pusha / pushad	1	60
popa / popad	1	61

图 5-7　push all 与 pop all 指令

练习 5.1

1. 对下面的每条指令，给出操作码和目标代码的字节数。假设 Double 是存储器中的双字。

   a. push ax　　b. pushd 10
   c. pushad　　d. pop ebx
   e. pop Double　　f. popad
   g. pushf

2. 对于下面的每个问题，假设给出指令“执行前”的值，计算“执行后”的值。画出堆栈示意图，跟踪指令执行过程。

指令执行之前	执行指令	指令执行之后
a. ESP: 06 00 10 00 ECX: 01 A2 5B 74	push ecx pushw 10	ESP, ECX
b. ESP: 02 00 0B 7C EBX: 12 34 56 78	pushd 20 push ebx	ESP, EBX
c. ESP: 00 10 F8 3A EAX: 12 34 56 78	push eax pushw 30 pop bx pop ecx	ESP, EAX, BX, ECX

3. 许多微处理器没有与 xchg 等价的指令。对于这类系统，下面的指令可用来交换两个寄存器的内容：

```
push  eax
push  ebx
pop   eax
pop   ebx
```

请解释为什么这些指令可以交换 EAX 和 EBX 寄存器的内容。比较执行这些指令与执行指令 xchg eax，ebx 所需代码的字节数。

4. 还有一种可选的 xchg 指令，其格式如下：

```
push  eax
mov   eax, ebx
pop   ebx
```

请解释为什么这些可以用来交换 EAX 和 EBX 寄存器的内容。比较执行这些指令与执行指令 xchg eax，ebx 所需代码的字节数。

5.2 过程体、调用及返回

过程在高级语言中可描述为一个自包含的子程序，主程序或者其他子程序通过语句——该语句包含了过程名以及与过程形参有关的参数列表能够调用过程。

许多高级语言将执行某个动作的过程和带有返回值的函数区分开来。一个**函数**与一个过程类似，但是，函数能够通过在表达式中使用函数名和参数表来调用。此外，它返回一个与函数名有关的值，并且这个值可以被表达式使用。从这种意义上讲，在 C/C++ 语言中，所有的子程序都是函数，而且允许函数没有返回值。

在汇编语言和一些高级语言中，术语过程用来描述两种类型的子程序，一种可以有返回值，一种可以没有返回值，本书中的过程基于上述两种类型。

同高级语言一样，过程在汇编语言中也是很有用的。过程不仅能将程序划分成多个容易处理的任务，而且可以分开写代码，这样可以在单个程序中多次使用这些代码，或者将它们保存起来，被其他程序重用。有时用汇编语言编写的代码比用高级语言编译器生成的代码的效率更高，且在任务运行不需要非常高效的情况下，这些代码可作为一个过程，可以被高级语言调用。

本节讨论如何编写80x86的过程，同时介绍如何使用微软的软件对它们进行编译与链接。主要内容包括如何定义一个过程，如何调用过程并返回。如何使用堆栈来保存寄存器的内容，这样，当过程返回到调用者时，几乎所有寄存器的内容都没有改变。还有一些与过程有关的重要概念需要考虑，包括如何传递参数给过程，以及如何在过程体中使用局部变量等，这些将在以后章节中讨论。

一个过程的代码总是出现在一条 .CODE 指令后。每一个过程体都包含两条指令：PROC 和 ENDP，每条指令都有一个标号来标识该过程的名字。用微软的宏汇编器，PROC 指令允许说明一些属性；这里只使用其中的一个，即 NEAR32。该属性表明过程位于和调用代码同样的代码段内，并且使用32位的地址，这对于平面的32位内存模式编程是正常选择。图5-8列出了包含过程 Initialize 的一个程序的相关部分，这个过程的主要工作是对几个变量进行初始化，这里只列出简要的程序调用，但是过程代码是完整的。

在图5-8中，过程 Initialize 以指示性指令 PROC 开始并以 ENDP 结束。属性 NEAR32 表明这是一个近程（near）过程。虽然这个例子的过程体在主程序代码前，但它也可放在主程序之后。回想一下，一个程序的执行并一定是从代码段的第一句开始，标号 _start 表示第一条要执行的指令。

过程 Initialize 的大部分语句是常用的 mov 指令。在这个主程序中，有两个地方用了 call 语句，使用过程可以使主程序代码更简短清晰。该过程影响了主程序数据段中定义的双字以及 EBX 寄存器，它没有局部变量。

当主程序执行时，指令

```
call Initialize
```

将控制从主程序转到过程。主程序两次调用该过程，通常，一个过程可以被调用任意次。返回指令

```
ret
```

把控制从过程返回给主程序调用者，在过程中至少会有一条 ret 指令，还可以有多条。如果只有一条 ret 指令，那么通常这条指令就是该过程的最后一条指令，因为没有这条指令的话，调用语句后的其他指令就无法执行。虽然一条 call 指令必须说明它的目的地址，但 ret 指令不需要，它会将控制转到最近的 call 指令后的那条指令。该指令的地址存储在80x86的堆栈中。

当对图5-8的示例程序进行汇编、链接和执行后，没有什么信息输出。然而，使用 Windbg 的工具跟踪执行的话，就可获得很多信息。图5-9显示 Windbg 的初始窗口。注意，ESP 的值为 0012ffc4。打开的存储器窗口是从地址 0012ffa4 处开始，即从栈顶开始的第32个字节。EIP 寄存器的值是 0040103e，这是要执行的第一条指令的地址（第一次调用）。图5-10是该调用执行之后的状态。现在 EIP 寄存器的值为 00401010，即过程 Initialize 中第一条语句的地址。把4个字节压入堆栈后，ESP 寄存器的值将为 0012ffc0。查看存储器中该地址的内容（图中突出显示部分），可以看到它的值为 43 10 40 00——也就是 00401043，这比第一次调用的地址增加了5个字节。如果检查程序的列表文件，可以发现每一条调用指令占用了5个字节的目标代码。因此，00401043 就是紧跟在第一条 call 指令后的指令的地址。

```
; procedure structure example
; Author:  R. Detmer
; Date:    revised 7/2005

.386
.MODEL FLAT

ExitProcess PROTO NEAR32 stdcall, dwExitCode:DWORD

.STACK  4096              ; reserve 4096-byte stack

.DATA                     ; reserve storage for data
Count1      DWORD   11111111h
Count2      DWORD   22222222h
Total1      DWORD   33333333h
Total2      DWORD   44444444h
;           other data here

.CODE                             ; program code

Initialize  PROC    NEAR32
            mov     Count1,0          ; zero first count
            mov     Count2,0          ; zero second count
            mov     Total1,0          ; zero first total
            mov     Total2,0          ; zero second total
            mov     ebx,0             ; zero balance
            ret                       ; return
Initialize  ENDP

_start:                               ; program entry point
            call    Initialize        ; initialize variables

; - other program tasks here

            call    Initialize        ; reinitialize variables

; - more program tasks here

            INVOKE ExitProcess, 0     ; exit with return code 0
PUBLIC _start                     ; make entry point public

END                               ; end of source code
```

图 5-8　过程结构

通常，call 指令会把下一条指令的地址（调用指令之后的第一条指令）入栈，然后转去调用过程代码。近程调用指令将 EIP 压入堆栈，然后改变 EIP 内容，让它指向被调用过程的第一条指令的地址。

从过程返回到主程序，其执行顺序与调用过程的顺序相反，ret 指令取出 EIP 中的数据，这样，下一条要执行的指令就是曾经压入堆栈保存地址的指令。

80x86 程序可以使用平面存储模式，也可用分段存储模式。使用分段存储模式，过程和调

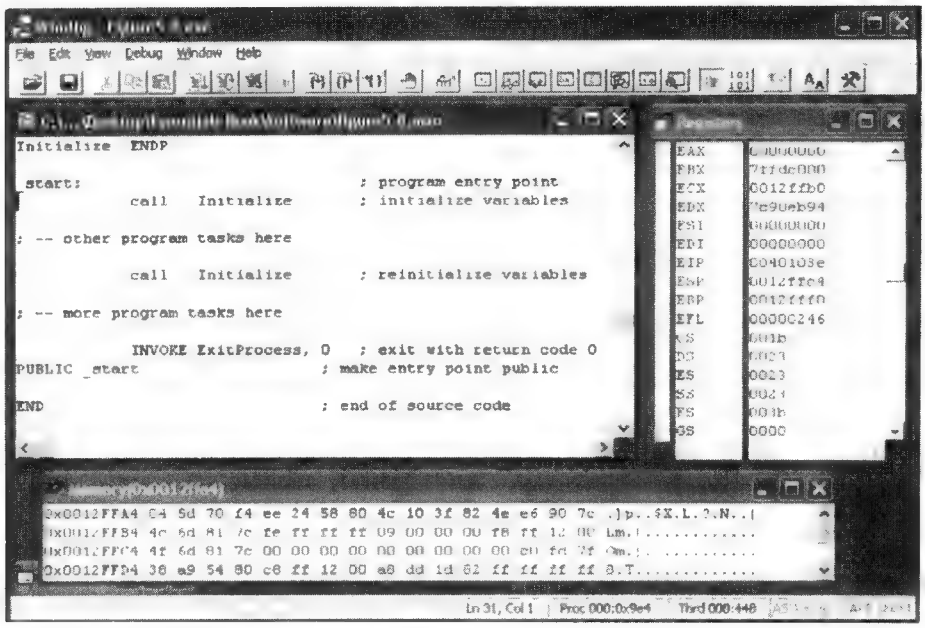


图 5-9 过程调用之前的状态

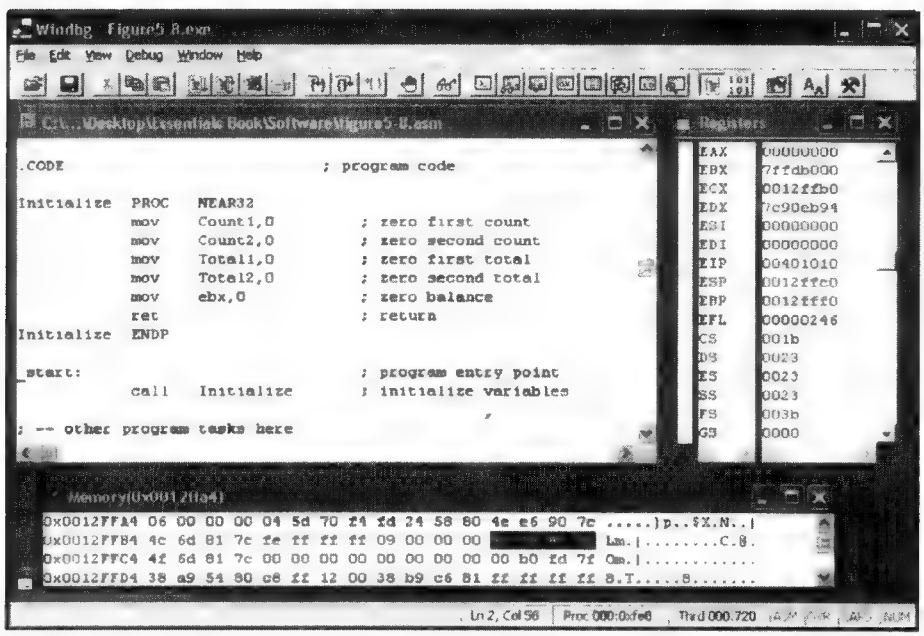


图 5-10 过程调用之后的状态

用它的程序代码可以不在同一个段内。事实上，对于 16 位分段编程，段长最大为 65 536 字节。所以，过程经常是放在不同的段内。80x86 体系结构采用远程（far）调用，转去调用不同内存段的过程：远程调用将 EIP 和 CS 压入堆栈，远程返回从堆栈中取出 EIP 和 CS。对于 32 位平面内存模式编程，一般采用近程调用。

80x86 调用指令的语法是：

```
call destination
```

图 5-11 列出了一些可用的 80x86 调用指令，图中省略了 16 位形式的调用和主要用于系统程序设计的调用。图 5-8 的程序包含了一个近程调用过程，该过程是由 PROC 操作数 NEAR32 标识。通常，通过 PROC 指令或其他指令或操作数，汇编器确定目标地址是否指向一个近程或远程过程，调用指令不修改任何标志位。

操作数	字节数	操作码
近程相对	5	E8
使用寄存器的近程间接	2	FF
使用存储器的近程间接	2+	FF
远程直接	7	9A
远程间接	6	FF

图 5-11　过程调用指令

本书中使用的全部过程都是第一种类型，即近程相对（near relative）。对于近程相对类型的过程，汇编器计算 32 位目标地址的偏移量，E8 操作码加上这个偏移量组成 5 个字节的指令。过程调用时的控制转移类似于相对跳转指令，但是，EIP 中原来数据要压入堆栈。

近程间接调用使用 32 位寄存器或存储器中的一个双字。当执行调用时，寄存器或双字的内容就是被调用过程的地址。这样，一条调用指令能够在不同时间调用不同的过程。

所有远程调用必须提供新的 CS 和 EIP 中的值，使用远程直接调用时，CS 和 EIP 中的值要写在指令中，这 6 个字节加上一个字节的操作码就是 7 个字节，如图 5-11 所示。对于远程间接调用，设置了一个 6 字节的内存块，该内存块的地址在指令中给出。增加的一个字节是 mod-reg-r/m 字节。

返回指令 ret 用于将控制从过程体返回到调用点。它的基本操作很简单，取出以前存放在堆栈中的地址，然后将该地址放入指令指针寄存器 EIP。由于堆栈保存了跟在调用指令后的下一条指令的地址，因此，程序会在该地址处继续执行。近程返回必须恢复 EIP 的内容，远程返回与远程调用步骤相反，恢复 EIP 和 CS 的内容，从堆栈中取出 EIP 和 CS 的值。

对于 ret 指令，有两种不同的格式。较简单的一种形式是没有操作数，其简单代码如下：

```
ret
```

还有一种形式是有一个操作数，代码如下：

```
ret count
```

在完成返回过程（恢复 EIP 内容，对于远程过程，还要恢复 CS 内容）的一些步骤后，操作数 count 加上 ESP 内容，并将结果存放到 ESP 中。对于过程调用而言，如果其他值（特别是参数）已经保存在堆栈里，这将是很有用的。在过程结束时，还要逻辑上清除这些值，从 ESP 中移出它们即可。参数将在下一节进一步讨论。图 5-12 列出了 ret 指令的各种格式。

如果过程的语句 PROC 有一个 NEAR32 的操作数，那么汇编器将为过程生成一个近程调用，

类型	操作数	字节数	操作码
近程	无	1	C3
近程	立即数	3	C2
远程	无	1	CB
远程	立即数	3	CA

图 5-12 ret 指令

并使用近程返回。Macro 汇编器还有用于强制近程或远程返回的 retn（近程返回）和 retf（远程返回）助记符，但这里没有用到这些助记符。

为大型程序创建内存区，需要汇编一个过程或者通过调用不同过程组成的过程组；也就是说，过程和调用程序可以在不同的文件中。这就需要一些额外的步骤。首先，必须汇编这些过程，这样，这些过程名就可在包含这些过程的文件的外部可见。其次，必须让调用程序知道外部过程的一些必要信息。最后，必须链接这些附加的 .OBJ 文件以得到一个可执行的程序。

PUBLIC 指示性语句用于让过程名在包含它们的文件外部可见，这与我们曾经用于使 _start 标号可见是一样的。通常，它的语法格式如下：

```
PUBLIC symbol1 [,symbol2]…
```

一个文件可能包含多条 PUBLIC 指示性语句。

EXTRN 指示性语句给调用程序提供有关外部标号（symbol）的信息。它可以有多项候选项，包括：

```
EXTRN symbol1:type [, symbol2:type]
```

一个文件可以包含多条 EXTRN 指示性语句。图 5-13 给出了两个过程 Procedure1 和 Procedure2 是如何一起汇编成一个独立于主程序代码的文件的。注意：这里需要 .386 和 .MODEL FLAT 指示性语句。

```
PUBLIC Procedure1, Procedure2
.CODE
Procedure1      PROC    NEAR32
                ...
Procedure1      ENDP
Procedure2      PROC    NEAR32
                ...
Procedure2      ENDP
                END
```

文件包含过程定义

```
EXTRN Procedure1:NEAR32, Procedure2:NEAR32
.CODE           ...
                ...
                call    Procedure1
                ...
                call    Procedure2
                ...
                END
```

文件包含过程调用

图 5-13 外部过程的代码

上述文件可像主程序一样汇编，每一次汇编产生一个 .OBJ 文件。为了链接这些文件，只要在 link 命令中列出所有的 .OBJ 文件——用已经单独汇编过的 kernel32.lib 文件来链接程序。

本节最后以一个计算正整数 Nbr 平方根的过程 Root 为例，该过程需要求出满足 SqRt*SqRt ≤ Nbr 的最大整数 SqRt。该过程代码如图 5-14 所示，这不是一个能够汇编的完整文件，但过程

的代码可以用图 5-13 的语句分别汇编，或者可以将它放在调用程序的文件中。

过程 Root 实现如下：

```
Sqrt := 0;
while Sqrt*Sqrt ≤ Nbr loop
        add 1 to Sqrt;
end while;
        subtract 1 from Sgrt;
```

```
; procedure to compute the integer square root of number Nbr
; Nbr is passed to the procedure in EAX.
; The square root SqRt is returned in EAX.
; Other registers are unchanged.
; author:  R. Detmer    revised:  08/2005

Root      PROC  NEAR32
          push  ebx           ; save registers
          push  ecx
          mov   ebx, 0        ; SqRt := 0
WhileLE:  mov   ecx, ebx      ; copy SqRt
          imul  ecx, ebx      ; SqRt*SqRt
          cmp   ecx, eax      ; SqRt*SqRt <= Nbr ?
          jnle  EndWhileLE    ; exit if not
          inc   ebx           ; add 1 to SqRt
          jmp   WhileLE       ; repeat
EndWhileLE:
          dec   ebx           ; subtract 1 from SqRt
          mov   eax, ebx      ; return SqRt in EAX
          pop   ecx           ; restore registers
          pop   ebx
          ret                 ; return
Root      ENDP
```

图 5-14　计算整数平方根过程

算法采用不断逼近整数 SqRt 的方法；当尝试值超过正确值时，就取上一个计算的结果作为最后的结果。这不是一个非常有效地算法，但它易于实现。

调用程序必须将 Nbr 的值放在 EAX 寄存器中，下一节将讨论一种更常用的方式来为过程传递参数。求平方根的过程 Root 将返回 SqRt 的值，并把它放在 EAX 寄存器中。由此可见，返回一个整型值的函数通常是用累加器实现。

除了实现该算法外，该过程在开始时还包含两条 push 指令，调用返回前，还有两条相应的 pop 指令。这些指令用于保存 EBX 和 ECX 寄存器的内容；也就是说，在调用过程 Root 前，将寄存器的原有数据返回给调用程序。这样，过程与调用程序相互独立，在使用过程 Root 时不必担心出现意外的结果，这一点将在下一节做进一步的讨论。

练习 5.2

1. 假设练习 1 中 NEAR32 过程被指令 call　Exercise1 调用，如果这个调用语句的地址是

00402000 且在调用之前 ESP 的值为 00406000。Excecisel 过程的第一个指令执行后，堆栈将返回什么地址？ESP 的值是多少？

2. 为什么分别编译过程时，要使用 PUBLIC 指令？为什么分别编译过程时，要使用 EXTRN 指令？
3. 当传递一个负数给过程 Root（图 5-14）时，过程将返回什么内容？

编程练习 5.2

1. 编写一个主程序，输入一个正整数，存储在 EAX 中，调用过程 Root（图 5-14），求该整数的平方根。要求主程序和过程 Root 在同一文件中，并且一起汇编。
2. 除了在不同的文件中分别汇编过程 Root 和主程序，重新设计编程练习 1。

5.3 参数与局部变量

在高级语言中，当过程被调用时，过程定义常常包括与变元（argument，有时称为实参）有关的参数（parameter，有时称为形参）。当过程被调用时，过程的（值传递）参数、实参值（可以是表达式），被复制到参数上。然后，这些值可被过程的局部变量（用于定义参数）所引用。输入输出（通过地址或变量传递）参数将一个参数标识符与一个单变量的变元关联在一起，该参数标识符在调用者与过程之间相互传递值。本节讨论了参数传递的常用方法，该方法可为输入参数传递字或双字型的值，或者在调用程序中为输入输出参数传递数据地址。

尽管简单的过程可以只用寄存器传递参数，但是大多数过程要用堆栈来传递参数。堆栈通常还用来存储局部变量，使用堆栈来传递参数与存储局部变量的技术密切相关。

下面通过一个简单的例子来说明堆栈是如何传递参数的。假定一个 NEAR32 的过程 Add2 的工作是将两个双字节整型数相加，并将相加的和返回到 EAX 寄存器。如果调用程序是通过将参数压入栈的方式来传递，那么它可以编码如下：

```
push    Value1      ; first argument value
push    ecx         ; second argument value
call    Add2        ; call procedure to find sum
add     esp,8       ; remove parameters from stack
```

在考虑如何从堆栈存取参数值之前，要注意的问题是，执行调用指令后，这些参数是如何从堆栈中取出的。这里不需要将从堆栈中弹出的参数放到某一目的地址，仅仅在堆栈指针上加 8，移走 ESP 之上的参数。从堆栈中移出参数很重要，因为重复的过程调用可能会耗尽堆栈空间。更严重的是，如果过程调用是嵌套的，并且内部调用在堆栈中留下了参数，那么外部返回在堆栈中将找不到正确的返回地址。一个可选的方法就是在调用程序的堆栈指针上加 *n*，在过程中使用 ret *n* 指令，在取出返回地址后，这种形式的返回指令将 ESP 内容加 *n*。这两种形式本书都会举例说明。

图 5-15 说明了过程 Add2 是如何将两个参数值从堆栈中取出的，过程代码使用**基地址模式**。在这种模式下，存储器地址是基地址寄存器的内容加上指令中的位移量的和。微软汇编器允许一个基地址用多种符号表示，本书使用 [寄存器＋数字] 的形式，例如，[ebp + 6]。任何通

用的寄存器（如 EAX、EBX、ECX、EDX、ESI、EDIEBP 或 ESP）都可以作为基地址寄存器，EBP 通常用于访问堆栈中的值。

```
Add2        PROC NEAR32   ; add two words passed on the stack
                          ; return the sum in the EAX register
            push   ebp                ; save EBP
            mov    ebp,esp            ; establish stack frame
            mov    eax,[ebp+8]        ; copy second parameter value
            add    eax,[ebp+12]       ; add first parameter value
            pop    ebp                ; restore EBP
            ret                       ; return
Add2        ENDP
```

图 5-15　堆栈中使用参数值传递

通过这种方式传递实参值的过程如下。在进入过程之前，堆栈中的内容如图 5-16 的左边所示。调用过程的指令为：

```
push    ebp                 ; save EBP
mov     ebp,esp             ; establish stack frame
```

这两条指令执行后，堆栈的内容如图 5-16 右边所示。

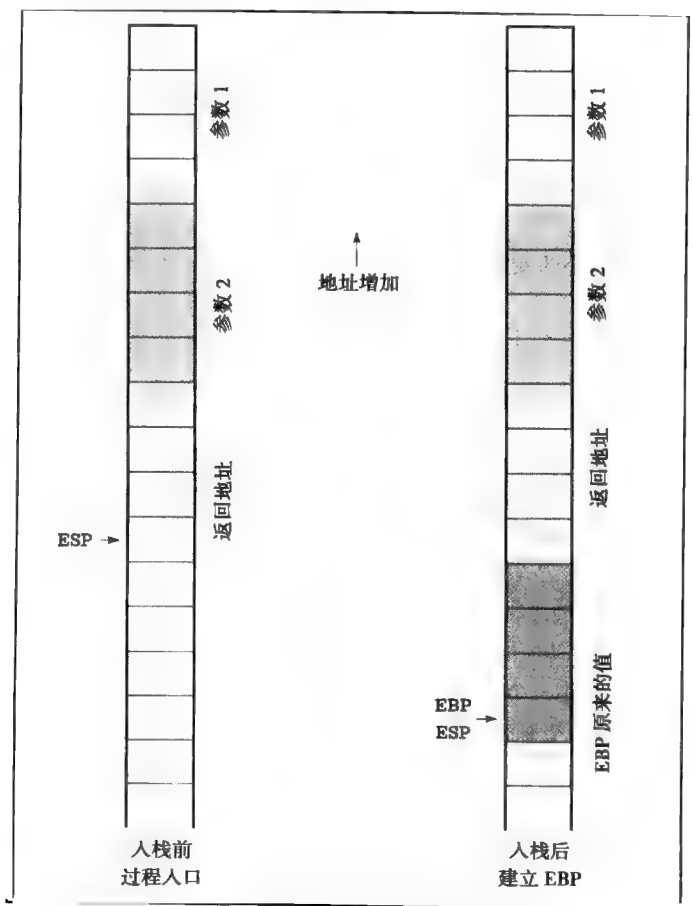


图 5-16　创建过程入口代码的基址指针

在EBP（以及ESP）中的地址和第二个参数值之间有8个字节的存储空间。因此，参数2的地址是[ebp+8]，第一个参数值在堆栈的高4字节，其地址为[ebp+12]。代码：

```
mov     eax,[ebp+8]          ; copy second parameter
add     eax,[ebp+12]         ; add first parameter
```

使用堆栈中存储器的值来计算最终的和。

这里为什么要使用EBP？为什么不只使用ESP作为基地址寄存器呢？其主要原因是ESP内容很可能会改变，但指令mov ebp，esp将堆栈中固定的参考指针放入EBP中。即使堆栈用于其他目的，如压入另外的寄存器值或调用其他过程，这个固定参考指针都不会改变。过程退出代码如下：

```
pop     ebp                 ; restore EBP
ret                         ; return
```

其中EBP的作用是恢复调用程序所使用的值。

有些过程需要为局部变量分配堆栈空间，并且大多数的过程需要保存寄存器的内容，正如图5-14所示。实现这些任务的指令和下面的两条指令一起组成一个过程的入口代码：

```
push    ebp                 ; save EBP
mov     ebp,esp             ; establish stack frame
```

但是，这两条指令是第一段入口代码指令。通过它们，可以计算最后的参数是存储在EBP的参考指针之上的8字节。EBP寄存器本身总是第一个压入，最后一个弹出，这样返回到调用程序的值和调用前的值相同。

现在考察堆栈是如何为局部变量分配空间的。首先回顾一下编程练习4.3中计算两个整数的最大公约数的算法。

```
gcd := number1;
remainder := number2;

until (remainder = 0) loop
      dividend := gcd;
      gcd := remainder;
      remainder := dividend mod gcd;
end until;
```

图5-17说明该设计是通过一个NEAR32的过程来实现，该设计实现了计算两个双字节整型数的最大公约数的算法，这两个整数值通过堆栈传递给过程，返回的最大公约数GCD将存放到EAX中，除了这个过程外，图5-17提供了一个完整文件，可以独立汇编。

该过程中，gcd被存储在堆栈中，直到返回值存入EAX寄存器中，指令

```
sub     esp,4        ; space for one local doubleword
```

```
PUBLIC GCD
; Procedure to compute the greatest common divisor of two
; doubleword-size integer parameters passed on the stack.
; The GCD is returned in EAX.
; No other register is changed.  Flags are unchanged.
; Author:  R. Detmer    Revised:  08/2005

GCD     PROC    NEAR32
        push    ebp             ; establish stack frame
        mov     ebp,esp
        sub     esp,4           ; space for one local doubleword
        push    edx             ; save EDX
        pushf                   ; save flags

        mov     eax,[ebp+12]; get Number1
        mov     [ebp-4],eax ; GCD := Number1
        mov     edx,[ebp+8] ; Remainder := Number2
until0: mov     eax,[ebp-4] ; Dividend := GCD
        mov     [ebp-4],edx ; GCD := Remainder
        mov     edx,0           ; extend Dividend to doubleword
        div     DWORD PTR [ebp-4]    ; Remainder in EDX
        cmp     edx, 0          ; remainder = 0?
        jnz     until0          ; repeat if not

        mov     eax,[ebp-4] ; copy GCD to EAX

        popf                    ; restore flags
        pop     edx             ; restore EDX
        mov     esp,ebp         ; restore ESP
        pop     ebp             ; restore EBP
        ret     8               ; return, discarding parameters
GCD     ENDP
        END
```

图 5-17　计算最大公约数过程

将堆栈指针向下移动 4 个字节，在存储 EBP 的位置下方，以及在存储其他寄存器的上方预留一个双字的空间，在 EDX 和标志寄存器压入堆栈后，堆栈的内容如图 5-18 所示，现在局部变量 gcd 可以在 [ebp-4] 地址处存取，这个地址是 EBP 中的固定参考指针下的 4 字节处。

该过程的其余部分的设计很容易实现。在这个例子中，一个寄存器可用于存储 gcd，但很多过程都有太多的局部变量，不能将它们都存储在寄存器中。因此，可以在堆栈中存储一些局部变量，在 [ebp-*offset*] 地址处存取。注意，在预留局部变量的空间后，要保存寄存器的内容，这样被存储的寄存器的数量不会影响变量的偏移量。还要注意的是，如果寄存器内容在返回调用程序后都没有改变，那么许多过程可保存超过两个寄存器的内容。

最后，讨论一下过程的退出代码：

```
popf                ; restore flags
pop     edx         ; restore EDX
```

```
mov     esp,ebp       ; restore ESP
pop     ebp           ; restore EBP
ret     8             ; return, discarding parameters
```

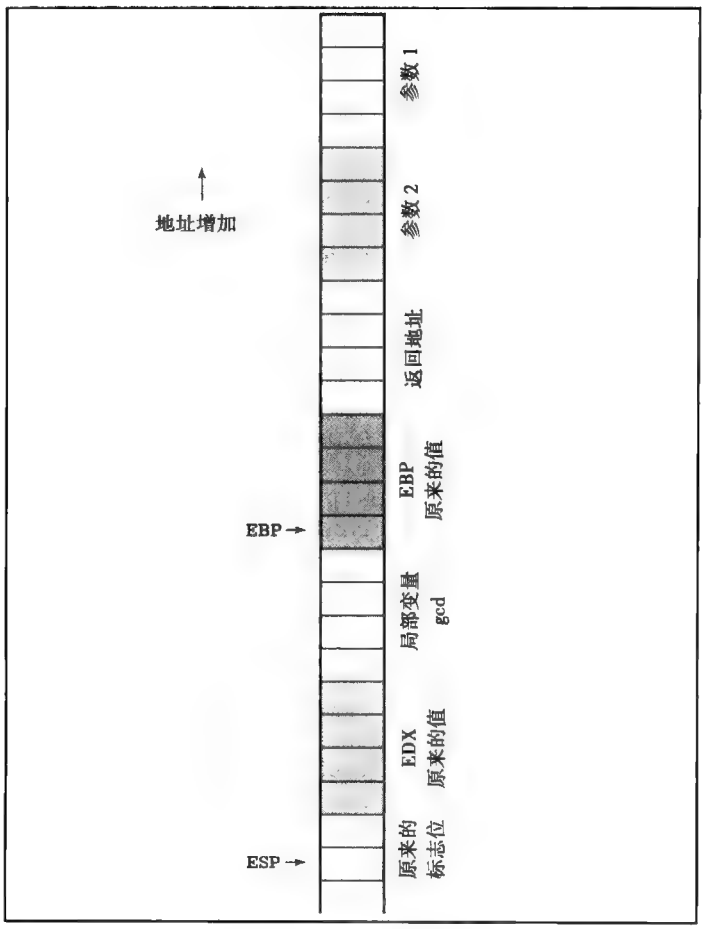


图 5-18 局部变量的栈使用

前两个 pop 指令是用来恢复标志寄存器和 EDX，它们出栈的顺序与入栈的顺序刚好相反。下一条指令应该是 add esp，4，它的作用是恢复进入代码中相应减法运算所造成的影响。但是，无论为局部变量分配多少的空间，指令 mov esp,ebp 都能更有效地做到这一点，而且像一条 add 指令一样，这条指令不会改变标志位。最后，使用了 ret 指令，操作数为 8，这样，对于这个过程而言，调用程序不必从堆栈中移出参数，这个任务由该过程完成。

图 5-19 总结了常用的过程进入和退出代码。高级语言的编辑器可为子程序产生类似的代码。实际上，通过这种方式可以编写能被高级语言程序所调用的汇编语言过程。编写过程的方法有很多，因此，在写过程之前，可以查看一下编译器的参考资料。

可能要注意的是，当过程返回一个值时，该值要存储在 EAX 中。对返回一个整数的函数而言，这是一种规定。

高级语言是如何用参数来返回值的呢？大量的参数（如：一个数组，一个字符串或者一条

记录）如何才能有效地传给一个过程？它们可以通过传递变量的地址而不是传递值给过程来实现。过程要么使用地址中的值，要么在这个地址存储一个新值。图 5-20 说明了一个使用 C++ 实现的过程，并用以下头部：

```
void Minimum(int A[], int Count, int& Min);
// Set Min to smallest value in A[0], A[1], ..., A[Count-1]
```

```
进入代码:
    push    ebp         ; establish stack frame
    mov     ebp,esp
    sub     esp,n       ; n bytes of local variables space
    push    ...         ; save registers
    ...
    push    ...
    pushf               ; save flags

退出代码:
    popf                ; restore flags
    pop     ...         ; restore registers
    ...
    pop     ...
    mov     esp,ebp     ; restore ESP if local variables used
    pop     ebp         ; restore EBP
    ret                 ; return
```

图 5-19　常用过程的入口和出口代码

在过程的实现中，对应于 A 和 Min 的实参地址传到了过程 Minimum。这个过程使用了寄存器间接寻址的方式，首先检查数组的每个元素，最后存储最小值。

指令 pushad 和 popad 用于保存和恢复所有通用寄存器的内容。这些指令的使用非常方便，但是如果过程产生的值要返回给寄存器时，它们就不能使用了。注意，由于参数 Count 是一个双字，第一个参数的地址（A 的地址）是在固定基址上面的 16 个字节，其中 EBP 占用 4 个字节，返回地址占用 4 个字节，Min 的地址占用 4 个字节，其余的 4 个字节是 Count 的值。（画出栈图。）

过程 Minimum 的调用代码如下：

```
lea     eax, Array      ; Parameter 1:  address of Array
push    eax
push    Count           ; Parameter 2:  value of Count
lea     eax, Min        ; Parameter 3:  address of Min
push    eax
call    Minimum         ; call procedure
add     esp, 12         ; discard parameters
```

这些代码执行之后，数组的最小值将存储在 Min 所指向的地址。

对于一个过程而言，将局部变量存在数据段中是十分合理的。.DATA 指示性语句可放在一

个由几个过程各自独立汇编的文件中。事实上，一个程序中可以有多条 .DATA 指示性语句，虽然通常没有必要这样做。变量可放在堆栈或者数据段部分，尽可能保持变量只在局部起作用，该数据段仅在包含定义的文件汇编时可见。即使过程与调用程序是在同一个文件中汇编，也应该避免在过程中直接引用调用代码的变量。

```
; Procedure to find the smallest doubleword in array A[1..Count]
; Parameters:  (1)  address of array A
;              (2)  value of Count (doubleword)
;              (3)  address of Min (destination for smallest)
; No register is changed.  Flags are unchanged
Minimum PROC   NEAR32
        push   ebp            ; establish stack frame
        mov    ebp,esp
        pushad                ; save all registers
        pushf                 ; save flags

        mov    ebx,[ebp+16]   ; get address of array A
        mov    ecx,[ebp+12]   ; get value of Count
        mov    eax,7fffffffh  ; smallest so far (largest positive integer)
        jecxz  endForCount    ; exit when no elements to check
forCount:
        cmp    [ebx],eax      ; element < smallest so far ?
        jnl    endIfLess      ; skip if not
        mov    eax,[ebx]      ; new smallest
endIfLess:
        add    ebx,4          ; address of next array element
        loop   forCount       ; iterate
endForCount:
        mov    ebx,[ebp+8]    ; get address of Min
        mov    [ebx],eax      ; move smallest to Min
        popf                  ; restore flags
        popad                 ; restore registers
        pop    ebp            ; restore EBP
        ret                   ; return
Minimum ENDP
```

图 5-20 使用地址参数的过程

值得注意的是，每个程序在退出时都有如下语句：

```
INVOKE ExitProcess, 0    ; exit with return code 0
```

INVOKE 并不是一条指令——MASM 相关文献中称它为指示性语句。然而，它更像是一个宏。实际上，如果指示性指令 .LISTALL 在上面这些代码之前使用，就可以得到一个扩展的代码：

```
push   +000000000h
call   ExitProcess
```

显然，这是用一个值为 0 的双字参数来调用过程 ExitProcess。

练习 5.3

1. 假设过程 NEAR32 开始为：

```
push  ebp          ; save EBP
mov   ebp,esp      ; new base pointer
push  ecx          ; save registers
push  esi
...
```

假设这个过程有 3 个参数传递到堆栈：一个双字，一个字，第二个字。画出上面的代码执行后的堆栈图，图中包括参数、返回地址以及 EBP 和 ESP 指向的字节。并说明每个参数是如何引用的。

2. 给出过程 NEAR32 的进入代码（图 5-19），该进入代码为局部变量预留了 8 个字节的堆栈存储空间，假设这个空间被 2 个双字占用，那么每个局部变量是如何引用的？
3. 解释为什么在将返回值放入寄存器 EAX 的过程中不能使用 pushad 和 popad 指令。

编程练习 5.3

写一个 NEAR32 过程，该过程实现下面具体的任务。对于每一个过程，使用堆栈传递参数给过程。除非具体说明要返回一个值放入寄存器中，否则寄存器中的值在过程中应该始终不变。也就是说，在过程中使用的寄存器（包括标志寄存器）应该在过程的开始保存起来，并在返回过程前恢复。根据局部变量分配堆栈空间，使用不带操作数的 ret 指令。对于下面的问题，分别编写一个汇编测试驱动程序，一个输入合适的值的简单的主程序，该主程序调用过程并在 Windbg 窗口中查看结果。主程序必须从堆栈中移出变量。链接和运行该程序。

1. 写一个过程 Min2，该过程找出两个双字长的整型参数中的最小值，并将这个最小值放在 EAX 寄存器中。
2. 写一个过程 Max3，该过程找出 3 个双字长的整型参数中的最大值。并将这个最大值放在 EAX 寄存器中。
3. 写一个过程 Avg，该过程找出一个数组中双字长的整数的平均值。过程 Avg 有 3 个参数：
   a. 数组的地址；
   b. 数组中整数的个数（以双字传递）；
   c. 用于存储结果的一个双字的地址。
4. 写一个过程 Search，该这程从双字长的数组中找到某个特定的值。如果在数组中找到这个值，返回这个值在数组中的位置（1，2，…，N），并将返回值放入 EAX 寄存器中，如果没有找到，则返回 0。过程 Search 有 3 个参数：
   a. 要搜索的值（一个双字）；
   b. 数组的地址；
   c. 数组中双字长的数的个数 N（以双字形式传递）。

递归的过程或函数是指直接或间接地调用它自己。下面两个练习要求使用递归函数，用 80x86 汇编语言编写一个递归程序几乎和编写普通程序一样简单。如果参数传到堆栈中，并且局部

变量也存在堆栈中，那么过程的每一个调用为其参数和局部变量分配新的存储空间。由于每个调用都有自己的堆栈，因此，传递给一个过程调用的参数就不会和其他调用混淆。如果寄存器的内容被保存和恢复，那么过程的每次调用都可使用同样的寄存器。

5. 阶乘函数定义如下，其中 n 是一个非负整数

$$factorial(n)=\begin{cases}1 & 当 n=0\\ n\times factorial(n-1) & 当 n>0\end{cases}$$

编写一个名为 Factorial 的过程，实现阶乘函数的递归定义。使用堆栈传递双字整型参数；函数返回的值放在 EAX 寄存器中。

6. 两个正整数 m 和 n 的最大公约数（GCD）可以通过下面伪代码所描述的函数来递归计算：

```
function GCD(m, n : integer) : integer;
if n = 0
then
    return m;
else
    Remainder := m mod n;
    return GCD(n, Remainder);
end if;
```

写一个能实现该递归定义的过程 GCD。使用堆栈来传递两个双字变量的值。返回函数值放在 EAX 寄存器中。

5.4 本章小结

本章讨论了在 80x86 体系中实现过程的相关技术。在过程的实现中，堆栈具有重要的作用。当过程被调用时，在控制转移给过程的第一条指令之前，下一条指令的地址会被存储在堆栈中。为了把控制返回给调用程序，返回指令必须从栈中取回该地址。参数值（或它们的地址）必须入栈以便能传给过程，入栈之后，基指针 EBP 和基地址为访问过程中的参数值提供了一种便捷的机制。堆栈可用来为过程的局部变量提供存储空间。堆栈总是用来“保存环境”，例如，当一个过程开始和在返回调用程序时，寄存器内容可被压入堆栈。这样，调用程序不必担心寄存器的内容会被过程改变。

第6章 位 运 算

一台计算机包含许多集成电路，这些电路能使计算机完成其功能。每个芯片都是由几个甚至几千个**逻辑门**组成，每个基本电路可执行由电子状态表示的位的**与**、**或**、**异或**、**非**等布尔运算，CPU 通常是 PC 中最复杂的集成电路。

前面的章节已经考察了一些 80x86 微处理器指令，其中包括数据传送、算术运算操作、分支和子程序调用等指令。80x86（以及其他大多数 CPU），一次也能执行多对位的布尔运算指令。本章定义了布尔运算，并详述了实现这些运算的 80x86 指令，包括引起位模式变化的指令，如在字节、字或双字中的移位和循环移位指令，或者从一个地址单元转移到另一个单元指令。虽然位运算指令非常简单，但是，由于它们能够提供一些在高级语言中很少用的控制；因此，在汇编语言设计中位运算指令被广泛应用。

6.1 逻辑运算

许多高级语言允许使用布尔类型的变量，也就是说，变量可以存储 true 或 false 值。实际上，所有的高级语言都允许在条件语句（if）中使用带布尔值的表达式。在汇编语言中，布尔值 true 用位值 1 表示，布尔值 false 用位值 0 表示。图 6-1 给出了用位值作为操作数的布尔运算的定义。或（or）运算有时称为“包含或”，以便与“异或”（xor）区分。or 和 xor 的唯一区别在于两个 1 运算时：1 or 1 得到 1；1 xor 1 得到 0。也就是说，异或值为 1 相当于两个操作数中有一个为真，但不是两个都为真。

bit1	bit2	bit1 and bit2
0	0	0
0	1	0
1	0	0
1	1	1

a）and 运算

bit1	bit2	bit1 or bit2
0	0	0
0	1	1
1	0	1
1	1	1

b）or 运算

bit1	bit2	bit1 xor bit2
0	0	0
0	1	1
1	0	1
1	1	0

c）xor 运算

bit	not bit
0	1
1	0

d）not 运算

图 6-1 逻辑运算的定义

80x86 用 and（与）、or（或）、xor（异或）和 not（非）指令来实现逻辑运算。这些指令的形式如下：

```
and   destination,source
or    destination,source
xor   destination,source
not   destination
```

前面三条指令可用于两个双字、字或者字节的操作数，对两个操作数相应的位进行逻辑运算。例如，当执行指令 and ebx，ecx 时，EBX 寄存器的第 0 位与 ECX 寄存器中的第 0 位进行“与”运算，EBX 寄存器的第 1 位与 ECX 寄存器中的第 1 位进行“与”运算，如此类推，直到 EBX 寄存器的第 31 位与 ECX 寄存器中的第 31 位进行“与”运算。所有 32 位数的“与”运算的结果按位存入目的地址。

not 指令只有一个操作数。它的作用是把操作数的每一位由 0 改为 1，由 1 改为 0。例如，如果 AH 寄存器存储的数为 10110110，那么执行 not ah 指令后，AH 寄存器中的内容变为 01001001。有时，not 运算也称为“对操作数取补”。

not 指令不影响任何标志位，但是，其他的三条布尔指令会影响 CF、OF、PF、SF、ZF 和 AF 等标志位。其中进位标志位 CF 和溢出标志位 OF 都置 0；辅助进位标志位 AF 可能被改变，但是没有定义；根据运算的结果，符号标志位 SF 和零标志位 ZF 置 1 或者置 0。例如，如果运算结果的每一位都是 0，那么 ZF 置 1；如果任意一位不为 0，那么 ZF 将置 0。奇偶校验位 PF 还没有用过，根据运算结果中低字节的奇偶性来对 PF 置 1 或置 0。

and、or 和 xor 指令都支持相同类型的操作数，并且需要相同字节的目标代码，图 6-2 对

目的操作数	源操作数	字节数	操作码		
			and	or	xor
8 位寄存器	8 位立即数	3	80	80	80
16 位寄存器	8 位立即数	3	83	83	83
32 位寄存器	8 位立即数	3	83	83	83
16 位寄存器	16 位立即数	4	81	81	81
32 位寄存器	32 位立即数	6	81	81	81
AL	8 位立即数	2	24	0C	34
AX	16 位立即数	3	25	0D	35
EAX	32 位立即数	5	25	0D	35
存储器字节	8 位立即数	3+	80	80	80
存储器字	8 位立即数	3+	83	83	83
存储器双字	8 位立即数	3+	83	83	83
存储器字	16 位立即数	4+	81	81	81
存储器双字	32 位立即数	6+	81	81	81
8 位寄存器	8 位寄存器	2	22	0A	32
16 位寄存器	16 位寄存器	2	23	0B	33
32 位寄存器	32 位寄存器	2	23	0B	33
8 位寄存器	存储器字节	2+	22	0A	32
16 位寄存器	存储器字	2+	23	0B	33
32 位寄存器	存储器双字	2+	23	0B	33
存储器字节	8 位寄存器	2+	20	08	30
存储器字	16 位寄存器	2+	21	09	31
存储器双字	32 位寄存器	2+	21	09	31

图 6-2 and、or 和 xor 指令

这些指令进行了总结。图 6-3 列出了 not 指令。

目的操作数	字节数	操作码
8 位寄存器	2	F6
16 位寄存器	2	F7
32 位寄存器	2	F7
存储器字节	2+	F6
存储器字	2+	F7
存储器双字	2+	F7

图 6-3 not 指令

值得注意的是，图 6-2 几乎与图 3-7 所描述的 add 和 sub 指令是一样的。同样，图 6-3 与图 3-10 所描述的 neg 指令相似。在这两种情况中，可使用的操作数类型相同，执行时间相同，甚至许多操作码也一样。回想一下，当操作码一样时，指令 add、sub、or 和 xor 中 mod-reg-r/m 字节的区别。图 3-8 说明了这些指令的 reg 字段。

下面通过一些例子来说明逻辑指令的工作过程。要计算结果，有必要把每个十六进制数扩展成二进制数，并对二进制数相应的位对进行逻辑运算，最后，把结果重新转换成十六进制。这些转换过程在例子中有相应的说明。为了使例子更简短，这里选择了一个字长的操作数。大多数的十六进制计算器可直接执行逻辑运算。

示例

指令执行前	指令	位运算	指令执行后
AX: E2 75 CX: A9 D7	and ax,cx	1110 0010 0111 0101 1010 1001 1101 0111 1010 0000 0101 0101	AX A0 55 SF 1 ZF 0
DX: E2 75 value: A9 D7	or dx,value	1110 0010 0111 0101 1010 1001 1101 0111 1110 1011 1111 0111	DX EB F7 SF 1 ZF 0
BX: E2 75	xor bx,0a9d7h	1110 0010 0111 0101 1010 1001 1101 0111 0100 1011 1010 0010	BX 4B A2 SF 0 ZF 0
AX: E2 75	not ax	1110 0010 0111 0101 0001 1101 1000 1010	AX 1D 8A

每条逻辑指令都有许多用途，and 指令应用之一是对目的操作数中选定的位清 0。注意：任意一位值和 1 进行“与”运算，其结果仍是原值；任意一位值与 0 进行“与”运算，其结果必定为 0。因此，要把一个字节或者字中的选定的位清 0，只要让它和一个要清 0 的对应位为 0、不需改变的位为 1 的数进行“与”运算就可以了。

例如，寄存器 EAX 内的数除最后四位外，将其余所有位清 0，可以用以下的指令

```
and   eax, 0000000fh      ; clear first 28 bits of EAX
```

如果 EAX 原来的值为 4C881D7B，它们之间的“与”运算的结果为 0000000B：

```
0100 1100 1000 1000 0001 1101 0111 1011 4C881D7B
0000 0000 0000 0000 0000 0000 0000 1111 0000000F
0000 0000 0000 0000 0000 0000 0000 1011 0000000B
```

在 0000000fh 的高位中只需一个 0 就可以，但是编写 7 个 0 有助于理解这个操作数的用途。尾部的十六进制数 f 对应的二进制数是 1111，所提供的四个 1 可以保证 EAX 的后四位保持不变。

在逻辑指令中，用于改变位值所对应的值通常称为**掩码**，微软的 MASM 编译器支持十进制、十六进制、二进制和八进制格式的数值。掩码通常使用十六进制和二进制表示，因为这样的位模式和二进制值对应，比较容易计算出相应的十六进制值。

如上所述，and 指令可将字节或字中的选定位清 0。在不改变其他位的情况下，or 指令可将一个字节或字中的选定位置 1。显然，1 无论是与 0 还是 1 进行 or 运算，结果还是 1；对于 or 运算，如果其中一个操作数为 0 时，or 运算的结果就是另一个操作数。

在不改变其他位的情况下，“异或”指令可对一个字节或字中选定的位按位取补。因为 0 xor 1 得到 1，1 xor 1 得到 0；也就是说，任一操作数与 1 进行 xor 运算，其结果是对该操作数取反。

逻辑指令的第 2 个应用是实现高级语言的布尔运算。存储器中的一个字节可以存放 8 个布尔值。设 flags 为这个字节，那么语句

```
and   flags, 11011101b    ; flag5 := false;  flag1 := false
```

将 false 赋值给第 1 位和第 5 位，而其他的位保持不变。（回想一下，位是从右到左进行编码，最右边的位是从第 0 位开始。）

如果存储器中 flags 字节被用于保存 8 个布尔值，那么，or 指令可以给它的任意选定的位赋值 true。例如，语句

```
or    flags, 00001100b    ; flag3 := true;  flag2 := true
```

将第 2 位与第 3 位设置为 true 值，而其他位保持不变。

如果存储器的 flags 字节被用于保存 8 个布尔值，那么 xor 指令能得到选定位的反。例如，设计语句

flag6 := NOT flag6;

可以通过下面的指令实现：

```
xor  flags, 01000000b      ; flag6 := not flag6
```

逻辑指令的第 3 个应用是实现一些算术运算，假定 EAX 寄存器中的值是一个无符号整数，值对 32 求余的表达式可以通过下面的指令计算。

```
mov   edx,0        ; extend value to quadword
mov   ebx,32       ; divisor
div   ebx          ; divide value by 32
```

按照这些指令，余数（值对 32 求余）将会保存在 EDX 寄存器，下面一种方法也可以把同样的结果放在 EDX 寄存器中，但是，商没有保存在 EAX 寄存器中。

```
mov   edx,eax          ; copy value to DX
and   edx,0000001fh    ; compute value mod 32
```

该方法比第一个方法更有效。因为 EDX 寄存器中的值是二进制数，计算如下：

$$bit31*2^{31} + bit30*2^{30} + . . . + bit2*2^2 + bit1*2 + bit0$$

因为从 $bit31*2^{31}$ 到 $bit5*2^5$ 的所有项都能被 32（2^5）整除，剩余部分被 32 整除的结果是尾部 5 位所表示的位模式，它可用 0000001F 进行掩码操作之后得到。其他相似的指令也可以这样计算，只要 mod 的第二个操作数是 2 的幂。

逻辑指令的第 4 个应用是处理 ASCII 码。回想一下，ASCII 码值为 30_{16} 表示 0，31_{16} 表示 1，依次类推，直到 39_{16} 表示 9。假定 AL 寄存器存放的是数字的 ASCII 码，且所对应的整数值是 EAX 寄存器所需的。如果 EAX 寄存器的高 24 位已知为 0，那么指令

```
sub   eax, 00000030h        ; convert ASCII code to integer
```

将实现把 ASCII 码转成整数。如果 EAX 中高位是未知的，那么指令

```
and   eax, 0000000fh        ; convert ASCII code to integer
```

是一种更安全的选择。它确保除 EAX 最后 4 位之外其余的位清 0。例如，如果 EAX 寄存器的值为 5C3DF036，去掉高位之后，AL 寄存器将保存字符 6 的 ASCII 码，那么执行指令 `and eax,0000000fh`，其结果 00000006 存入 EAX。

`or` 指令可以用来把寄存器中从 0 到 9 的整数转换成相应的 ASCII 码。例如，如果整数是在 BL 寄存器中，那么下面的指令将把 BL 寄存器的内容转换成相应的 ASCII 码：

```
or   bl,30h       ; convert digit to ASCII code
```

如果 BL 包含 04，那么 `or` 指令的结果是十六进制数 34：

```
0000 0100     04
0011 0000     30
0011 0100     34
```

在 80x86 处理器中，指令 `add bl, 30h` 可以用同样的时钟数周期数和目标代码完成 `or`

指令的上述功能。但是，对有些CPU而言，or运算比加法指令的效率更高。

xor指令可用来改变ASCII码字母的大小写。假定CL寄存器包含一些大小写字母的ASCII码，大写字母的ASCII码与小写字母的ASCII码只是在第5位上有差别。例如，大写字母S的编码是53_{16}（01010011_2）而小写字母s的ASCII码是73_{16}（01110011_2）。指令

```
xor   cl, 00100000b      ; change case of letter in CL
```

把寄存器CL内的第5位取反，从而改变ASC II码的值以进行大小写的转换。

80x86的指令集包括test指令，test指令除了不改变目的操作数之外，它的功能与and指令相同。也就是说，指令test唯一的任务是设置标志位（要记住，cmp指令实质上相当于设置标志位的sub指令，但是cmp指令不改变目的操作数）。test指令的应用之一是检查一个字节或字的特定位的值。下面的指令可测试DX寄存器的第13位的值。

```
test   dx, 2000h    ; check bit 13
```

值得一提的是，十六进制的2000与二进制的0010 0000 0000 0000是等价的，其中第13位等于1。通常，test指令后面都会跟随jz或者jnz指令，它的作用是根据第13位是0还是1，跳转到相应的目的操作数。

test指令也用来获取寄存器中值的信息，例如：

```
test   ecx, ecx     ; set flags for value in ECX
```

就是使寄存器ECX与自己进行and操作，其结果为原值（任意位和它自己执行and操作的结果还是它自己）。标志位的设置取决于ECX的值。指令

```
and   ecx, ecx      ; set flags for value in ECX
```

可完成相同的功能，并且效率也相当。但是使用test可以使指令的作用一目了然，即，仅仅用于测试。

test指令的各种形式在图6-4中列出。它们基本上与and、or和xor指令一样。当源操作数在存储器时，目的操作数只能是累加器，但是，MASM可指定任何一个寄存器作为目的操作数，并且MASM调换操作数，可将存储器操作数作为目的操作数，这也是一种允许的形式。

目的操作数	源操作数	字节数	操作码
8位寄存器	8位立即数	3	F6
16位寄存器	16位立即数	4	F7
32位寄存器	32位立即数	6	F7
AL	8位立即数	2	A8
AX	16位立即数	3	A9
EAX	32位立即数	5	A9
存储器字节	8位立即数	3+	F6
存储器字	16位立即数	4+	F7

图6-4 test指令

目的操作数	源操作数	字节数	操作码
存储器双字节	32 位立即数	6+	F7
8 位寄存器	8 位立即数	2	84
16 位寄存器	16 位立即数	2	85
32 位寄存器	32 位立即数	2	85
存储器字节	8 位立即数	2+	84
存储器字	16 位立即数	2+	85
存储器双字节	32 位立即数	2+	85

图 6-4 （续）

练习 6.1

1. 对于下面的每个问题，假定给出指令执行前的值，求指令执行后的值。

指令执行之前	执行指令	指令执行之后
a. BX: FA 75		
CX: 31 02	and bx,cx	BX, SF, ZF
b. BX FA 75		
CX 31 02	or bx,cx	BX, SF, ZF
c. BX FA 75		
CX 31 02	xor bx,cx	BX, SF, ZF
d. BX FA 75	not bx	BX
e. AX FA 75	and ax,000fh	AX, SF, ZF
f. AX FA 75	or ax,0fff0h	AX, SF, ZF
g. AX FA 75	xor ax,0ffffh	AX, SF, ZF
h. AX FA 75	test ax,0004h	AX, SF, ZF

2. 当 EAX 寄存器中 value 是无符号整数时，书中给出了 value 模 32 的两种计算方法：

```
mov   edx,0        ; extend value to quadword
mov   ebx,32       ; divisor
div   ebx          ; divide value by 32
```

与

```
mov   edx,eax          ; copy value to DX
and   edx,0000001fh    ; compute value mod 32
```

计算每种方法所需目标代码总的字节数。

3. 假定 EAX 寄存器中的 value 是无符号整数。给出合适的指令，计算 value 模 8，并把结果保存在 EBX 寄存器，保持 EAX 寄存器的内容不变。

4. 假定双字长 flags 的每一位表示一个布尔值，0 位对应 flag0，依此类推，31 位对应 flag31。根据下面每条语句，给出一条 80x86 指令实现这条语句：

   a. flag2 :=true;

   b. flag5 :=false; flag16 :=false; falg19 :=false;

   c. flag12 :=NOT flag12

5. a. 假定 AL 寄存器包含的是大写字母的 ASCII 码，给出能把它的内容转换成相应小写字母的逻辑指令（除 xor 之外）。

b. 假定 AL 寄存器的值是小写字母的 ASCII 码，给出能把它的内容转换成大写字母的逻辑指令（除 xor 之外）。

编程练习 6.1

1. 编写一个 NEAR32 的过程 atoi，该过程将数字的 ASCII 码字符串转换成双字整数。即，atoi 有一个参数，该参数是字符串在存储器中的地址；atoi 将从栈中移出这个参数。保留标志位和除 EAX 之外其他寄存器的内容不变。通过下面的算法，该过程返回一个值，并存放在 EAX 寄存器中。

```
value :=0;
point at first character;
while character is the ASCII code for a digit loop
    find digitValue that corresponds to character;
    value := 10*value + digitValue;
    point at next character;
end while;
return value;
```

该过程可用来把整数形式转换成可用于计算的二进制补码形式。通常输入的字符串是以空字符 null（00）结束，但是该设计一旦发现非数字字符时，将终止遍历存储器。假定传给 atoi 过程的字符串是 35 37 30 00，它返回值 0000023A 并存放在 EAX 中。通过一个调用它的主程序来测试所编写的过程。测试驱动器将在数据段中声明要转换的字符串。

2. 修改过程 atoi（编程练习 1），实现跳过输入字符中开始的空格（尾部空格将终止输入）
3. 编写一个 NEAR32 的过程 itoa，该过程将双字的非负整数转换为 ASCII 码字符所表示的数字。即，itoa 将有两个参数：（1）双字整数，（2）存储器中 10 字节长字符串的地址。itoa 将从栈中移出这些参数。保留标志位和除 EAX 之外其他寄存器的内容不变。通过下面的算法，该过程返回一个值，并存放在 EAX 寄存器中。

```
for index := 9 downto 0 loop
    divide value by 10, producing quotient and remainder;
    convert remainder to corresponding ASCII code and store at address+index;
    value := quotient;
end for;
```

该过程可用来把非负 2 进制补码整数转换成可读的形式。假定这个值是 000023A，那么产生的字符串将是 30 30 30 30 30 30 30 35 37 30，也就是“0000000570”。（为什么目的地址需要 10 字节长？）通过一个调用它的主程序来测试所编写的过程。主程序将在数据段中分配 10 字节长的空间来存放结果。

4. 修改编程练习 3 中的过程 itoa，使它也能处理负数，输出的字符串将是 11 位长，负数的最高位用“−”表示，而正数用“+”表示。
5. Pascal 程序设计语言包含了预定义函数 odd，它有一个双字整数参数和返回值，如果是奇数时，

该返回值为 true，而为偶数时返回值为 false。用汇编语言写一个 NEAR32 的过程来实现这个功能。在 EAX 中返回 −1 表示 true，0 表示 false。除了 EAX 寄存器之外，过程不能改变其他寄存器的内容。使用适当的逻辑指令产生返回值，该过程必须能够取出堆栈中的参数。保留标志位和除 EAX 之外其他寄存器的内容不变。通过下面的算法，该过程返回一个值，并存放在 EAX 寄存器中。

6. 用二维平面制图法设计一个平面矩形区域显示在显示器上，区域外的点要忽略掉。用 $x=x_{max}$，$x=x_{min}$，$y=y_{min}$，$y=y_{max}$ 这四条线来把这块区域划分如下：

0110	0100	0101	
0010	0000	0001	$y=y_{max}$
1010	1000	1001	$y=y_{min}$
$x=x_{min}$		$x=x_{max}$	

图中每个（x，y）坐标点对应一个 outcode（或者说区域码）。这个四位码按以下规则赋值：

- 如果这个点是在区域的右边，那么第 0 位（最右边）是 1，即 $x > x_{max}$，否则为 0。
- 如果这个点是在区域的左边，那么第 1 位（$x < x_{min}$）是 1。
- 如果这个点是在区域的上边，那么第 2 位（$y > y_{max}$）是 1。
- 如果这个点是在区域的下边，那么第 3 位（$y < y_{min}$）是 1。

上图说明了平面中每个区域的区域码。

a. 假设坐标点（x_1，y_1）的区域码放在 AL 的低四位，坐标点（x_2，y_2）的区域码放在 BL 的低四位，寄存器的其他位都为 0。给出一条 80x86 语句，使得当这两个坐标点都在区域内时，设置 ZF 为 1，否则 ZF 为 0，AL 或 BL 中的值可以改变。

b. 假设坐标点（x_1，y_1）的区域码放在 AL 的低四位，坐标点（x_2，y_2）的区域码放在 BL 的低四位，寄存器的其他位都为 0。给出一条 80x86 语句，使得当这两个点都在矩形区域的同一边时，设置 ZF 为 0（“在同一边”意味着，都在 $x=x_{max}$ 的右边，或都在 $x=x_{min}$ 的左边，或都在 $y=y_{max}$ 上边，或都在 $y=y_{min}$ 下边）。AL 和 BL 中的值可以改变。

c. 编写一个 NEAR32 过程 setcode，把区域码还原成坐标点（x，y）。特别是，过程 setcode 有 6 字长的整型参数：x、y、x_{max}、x_{min}、y_{max} 和 y_{min}。这些参数按给定顺序入栈。返回区域码放入寄存器 AL 中低四位，将 0 赋给 EAX 中的高位。保留标志位和除 EAX 之外其他寄存器的内容不变。通过一个调用它的主程序来测试所编写的过程。

7. 编写一个 NEAR32 的过程 hexToInt，它有一个参数需要传给堆栈，一个字符串的地址。atoi 将从栈中取出这个参数。这个过程与 atoi（编程练习 1）类似，只是这个过程将十六进制形式的数字字符串转换成一个双字，保存在 EAX 寄存器中。该过程跳过字符串开始的空格，并累加值，直到遇到的字符不是十六进制数为止（有效的字符是从 0 到 9，A 到 F，以及 a 到 f）。保留标志位和除 EAX 之外其他寄存器的内容不变。

6.2 移位与循环移位指令

汇编语言编程人员使用上一节介绍的逻辑指令能够设置或清除寄存器或存储器中的字或字节的位；使用移位和循环移位指令能够改变一个双字、字或者字节内位的位置。本节详细介绍

了移位和循环移位指令，并给出了一些应用的实例。

移位指令对目的操作数所给定单元内的位进行左移或右移。移位的方向是由指令助记符的最后一位决定的，sal 和 shl 是左移；sar 和 shr 是右移。移位指令可分为两类：逻辑移位指令和算术移位指令，shl 和 shr 是逻辑移位指令，sal 和 sar 是算术移位指令。下面将解释算术移位和逻辑移位的区别，图 6-5 列出了移位指令的助记符。

移位指令的格式如下：

```
s-      destination, count
```

操作数 count 有三种形式。这个操作数可以是数字 1，可以是一个字节的立即操作数，还可以是指定的寄存器 CL。早期 8086/8088 CPU 只有第一和第三这两种形式。

如果指令格式如下：

```
s-      destination, 1
```

那么，其作用是使目的操作数移动 1 位。而指令

```
s-      destination,   immediate8
```

可以对 0 到 255 之间的一个立即操作数编码。然而，大多数的 80x86 体系是通过 00011111_2 对该操作数进行掩码，也就是说，在移位前把这个操作数对 32 取模。这是因为对一个不超过一个双字长的操作数来说，做超过 32 位的移位操作是没有意义的。最后一种形式的移位指令为：

```
s-      destination,  cl
```

寄存器 CL 中是无符号计数操作数。同样，对于大多数 80x86 CPU 而言，在移位前先把计数操作数对 32 取模。

	左移	右移
逻辑	shl	shr
算术	sal	sar

图 6-5 移位指令

算术左移和逻辑左移的功能是一样的，助记符 sal 和 shl 产生相同的目标代码。当进行左移操作时，目的操作数的每一位向左移动，最右边的一位置 0。移出左边的位被丢失，最后移出的一位除外，它被保存在进位标志位 CF 中。根据目的地址中最后的结果，符号标志位 SF、零标志位 ZF 和奇偶校验标志位 PF 将赋相应的值。在多位移位中，溢出标志位 OF 未定义；在一位移位（count = 1）中，如果结果的符号位的值和原操作数符号位的值相同，则 OF 置为 0，反之，则置 1。辅助进位标志位 AF 未定义。

算术右移和逻辑右移不一样。对这两者而言，目的操作数的每一位向右移动，除了最后移出右边的一位被保存在 CF 中外，其余移出右边的位被丢失。对于逻辑右移（shr），最左边的一位置 0。而对于算术右移（sar），最左边的一位是原来的操作数的符号位填充。因此，对算术右移而言，如果原来的操作数是负的二进制补码数，那么新的操作数中最左边将一直用 1 填充，并且新的操作数仍为负。和左移一样，算术右移后标志位 SF、ZF 和 PF 的值取决于运算的结果，

AF 未定义。多位移位中，溢出标志位 OF 未定义。对于一位逻辑右移 shr，如果结果的符号位和原操作数的符号位的值相同，OF 置 0；反之，则置 1（注意：这相当于把原操作数的符号位的值赋给 OF）。对于一位算术右移 sar，OF 置 0，因为原来的操作数和新的操作数的符号位通常是相同的。

有些十六进制的计算器能直接进行移位运算。为了便于运算，需要使用二进制来编写操作数，移位或重组位（适当地用 0 和 1 填充），并把新的位模式转换回十六进制。对于 4 位或 4 的倍数的多位移位，问题的处理会简单一些。这种情况下，每 4 位一组对应一个十六进制，所以可以考虑对十六进制数而不是位进行移位。下面举例说明移位指令的执行过程，每一个例子的第一个字都是十六进制值 A9 D7（二进制 1010 1001 1101 0111）。移出的位用一条垂直线和原值分开，新增的位在新值中用加粗字体表示。

示例

指令执行前	指令	二进制运算	指令执行后
CX: A9 D7	sal cx,1	1|010 1001 1101 0111 ← 0101 0011 1010 111**0**	CX 53 AE SF 0 ZF 0 CF 1 OF 1
AX: A9 D7	shr ax,1	1010 1001 1101 011|1 → **0**101 0100 1110 1011	AX 54 EB SF 0 ZF 0 CF 1 OF 1
BX: A9 D7	sar bx,1	1010 1001 1101 011|1 → **1**101 0100 1110 1011	BX D4 EB SF 1 ZF 0 CF 1 OF 0
word at ace: A9 D7	sal ace,4	1010 |1001 1101 0111 ← 1001 1101 0111 **0000**	ace 9D 70 SF 1 ZF 0 CF 0 OF ?
DX: A9 D7	shr dx,4	1010 1001 1101|0111 → **0000** 1010 1001 1101	DX 0A 9D SF 0 ZF 0 CF 0 OF ?
AX: A9 D7 CL: 04	sar ax,cl	1010 1001 1101|0111 → **1111** 1010 1001 1101	AX FA 9D SF 1 ZF 0 CF 0 OF ?

图 6-6 给出移位指令中各种类型操作数所需的操作码和字节数。到目前为止，4 种类型的移位指令都已经讨论了，接下来要讨论的循环移位指令和前面讨论的这 4 种类型的移位指令共用操作码。目的操作数和计数操作数的类型隐含在操作码中。正如其他指令，目标代码 mod-reg-r/m 字节的 reg 字段被用来选择移位运算或循环移位运算的不同类型，以及运算是否在寄存器和存储器之间进行。

目的操作数	源操作数	字节数	操作码
8 位寄存器	1	2	D0
16/32 位寄存器	1	2	D1
存储器字节	1	2+	D0
存储器字 / 双字	1	2+	D1
8 位寄存器	8 位立即数	3	C0
16/32 位寄存器	8 位立即数	3	C1
存储器字节	8 位立即数	3+	C0
存储器字 / 双字	8 位立即数	3+	C1
8 位寄存器	CL	2	D2
16/32 位寄存器	CL	2	D3
存储器字	CL	2+	D2
存储器字 / 双字	CL	2+	D3

图 6-6 移位和循环移位指令

移位指令很简单，但是它们应用广泛。应用之一是进行乘法和除法运算。实际上，对于没有乘法指令的处理器，移位指令是做乘法运算的子程序的一个关键部分。即使对 80x86 体系结构而言，有些乘法计算用移位运算比用乘法指令更快。

在一个乘数为 2 的乘法运算中，被乘数左移一位的结果作为乘积放在初始单元，该乘积是正确的，除非进位标志位 OF 设置为 1。显然，该运算对无符号数有效；每位左移一位，新的数就是二进制表示的原数的 2 的高次幂，一位左移等于一个有符号操作数乘 2。实际上，在十六进制计算器中，一位左移的结果可用乘 2 来得到。

一位右移可用于计算一个无符号操作数除 2。例如，寄存器 EBX 内有一个无符号操作数，逻辑右移 `shr ebx, 1` 把 EBX 内的每一位移到相应的 2 的下一个低次幂，得到原来值的一半。初始单元的位复制到进位标志位 CF，这就是除法的余数。

如果 EBX 内有一个有符号操作数，那么算术右移指令 `sar ebx, 1` 和 `idiv` 指令在除数为 2 时结果几乎是一样的。不同的是，如果被除数是一个负的奇数，商就被取整；就是说，右移得出的值可能比用 `idiv` 指令得出的值要小。举一个具体的例子来说，假设寄存器 EDX 内容为 FFFFFFFF，寄存器 EAX 内容为 FFFFFFF7，那么 EDX:EAX 表示是 4 字长的二进制补码的 −9。同时，假定有 ECX 的内容为 00000002。那么 `idiv ecx` 得到的结果在 EAX 中为 FFFFFFFC，在 EDX 中为 FFFFFFFF；也就是说，商为 −4，余数为 −1。然而，如果把 FFFFFFF7 放在 EBX 中，`sar ebx,1` 得到的结果是 EBX 内容为 FFFFFFFB，CF 内容为 1，商为 −5，余数为 +1。其中，商和余数的关系满足下面的等式：

被除数＝商 × 除数＋余数

但是当商为 −5，余数为＋ 1 时，余数的符号和被除数的符号是相反的，这违反了 idiv 指令的规则。

一个操作数乘 2，可以用自己加上自己的加法运算实现，也可以用左移位运算实现。有时，移位运算的效率比加法高一些。不过，这两种运算的效率都比乘法高得多。一个操作数除 2，右移位是替代除法的唯一选择，而且速度更快；但是当被除数是负数时，对于除 2 运算，右移位和除法有很大的不同。无论是一个操作数乘或者除以 4、8 以及其他 2 的幂，都可以用重复执行一位移位指令或用一条多位移位指令来实现。

移位可以与其他的逻辑指令一起使用，把不同组的位结合成一个字节、字或双字，或者把一个字节、字或双字中的位分隔成不同的组。图 6-7 是一个名为 hexToAscii 的过程，它可以把

```
; test driver for hexToAscii procedure
.386
.MODEL FLAT
ExitProcess PROTO NEAR32 stdcall, dwExitCode:DWORD
.STACK  4096
.DATA
source  DWORD   0b70589a4h
dest    BYTE    8 DUP (?)
.CODE
_start:     push    source          ; operand to convert to hex
            lea     eax, dest       ; get destination address
            push    eax             ; second operand
            call    hexToAscii      ; call conversion procedure
            INVOKE  ExitProcess, 0  ; exit with return code 0
PUBLIC _start

hexToAscii  PROC    NEAR32
; Convert doubleword to string of eight ASCII codes
; Parameters
; (1) doubleword containing source value
; (2) doubleword containing destination address (address of 8-byte-long
area)
; Parameters are removed from the stack by this procedure
; Author:  R. Detmer
; Date:    08/2005

            push    ebp             ; establish stack frame
            mov     ebp, esp
            push    eax             ; save registers
            push    ebx
            push    ecx
            push    edx
            pushf                   ; save flags
            mov     eax, [ebp+12]   ; source value to EAX
            mov     edx, [ebp+8]    ; destination address to EDX
            mov     ecx, 8          ; loop count
forIndex:   mov     ebx, eax        ; copy value
            and     ebx, 0000000fh  ; mask off all but last 4 bits
ifDigit:    cmp     bl, 9           ; digit <= 9?
            jnle    elseHex         ; skip if not
            or      bl, 30h         ; convert digit to ASCII
            jmp     endIfDigit      ; exit if
elseHex:    add     bl, 'A'-10      ; convert letter to ASCII
endIfDigit:
            mov     [edx+ecx-1], bl ; store in memory
```

图 6-7 转换双字为十六进制的过程

```
            shr     eax, 4          ; shift next digit to right
            loop    forIndex        ; iterate loop

            popf                    ; restore flags
            pop     edx             ; restore registers
            pop     ecx
            pop     ebx
            pop     eax
            pop     ebp             ; restore base pointer
            ret     8               ; return, discarding parameters
hexToAscii  ENDP
END
```

图 6-7 （续）

一个双字操作数转换成存储器中一系列 8 字节的操作数，每个字节包含操作数中十六进制数所对应的 ASCII 码。例如，如果操作数是 B70589A4，那么过程将产生 42 37 30 35 38 39 41 34。测试驱动程序源代码与过程如图 6-8 所示。图 6-8 所示的屏幕截图显示了要退出时的测试程序。在存储器中可以看到突出显示的结果。

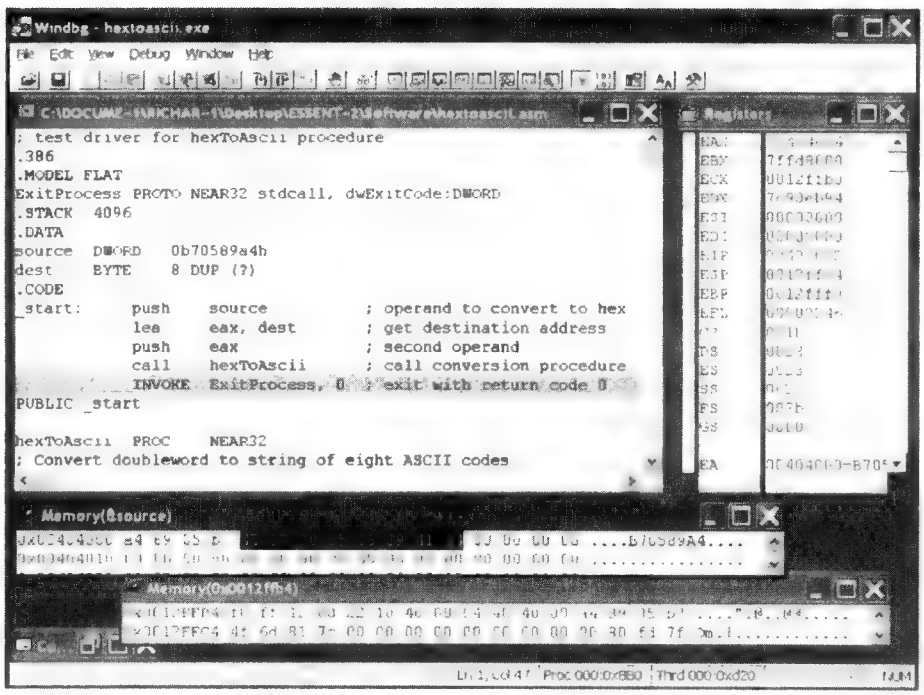


图 6-8 调用过程 hexToAscii 的结果

为了实现该功能，hexToAscii 必须从操作数中提取 8 组 4 位数的内容。每组表示从 0 到 15 的十进制数，并且每组要转换成相应的 ASCII 码。这些字符是数字 0 到 9，表示整数 0（0000_2）到 9（1001_2），或是字母 A 到 F，表示整数 10（1010_2）到 15（1111_2）。

生成的 8 个字符按照从右到左的顺序保存在存储器中的相邻字节。操作数最初被复制到 EAX，并重复地右移来获取右端的下一个四位。程序中间部分的设计如下：

```
for index := 8 downto 1 loop
    copy value to EBX;
    mask off all but last 4 bits in EBX;
    if value in BL ≤ 9
    then
        convert value in BL to a character 0 through 9;
    else
        convert value in BL to a letter A through F;
    end if;
    store BL in memory at address destination + index - 1;
    shift value right four bits to position next group of 4 bits;
end for;
```

在代码中，指令

```
and  ebx, 0000000fh  ; mask off all but last 4 bits
```

除了最后四位，屏蔽 EDX 中其他所有位。if 语句通过下面的代码来实现：

```
ifDigit:    cmp     bl, 9           ; digit <= 9?
            jnle    elseHex         ; skip if not
            or      bl, 30h         ; convert digit to ASCII
            jmp     endIfDigit      ; exit if
elseHex:    add     bl, 'A'-10      ; convert letter to ASCII
endIfDigit:
```

通过使用 or 指令可把 0 到 9 的数字转换成相应的 ASCII 码，这里可使用 add edx,30h 来实现该功能。为了把数字 0A 到 0F 转换为字母 A 到 F 对应的 ASC II 码 41 到 46，每个数字必须加上 'A' -10。这实际上是加了十进制的 55，但是这样写比使用 add edx,55 语句更直观。指令 shr 把寄存器 EAX 内的值右移四位，移出的十六进制数正好转换为一个字符。

循环移位指令与移位指令很类似，在移位指令中，从一端移出的位会被丢弃，而另一端空出的位会用 0 填充（对于负数的算术右移，用 1 填充）。而循环移位指令中，一端移出的位用来填充另一端空出的位。

循环移位指令的格式与单移位指令的格式相同。一位循环移位指令的格式是：

```
r-    destination,1
```

对多位循环移位，有两种形式：

```
r-    destination, immediate8
r-    destination,cl
```

rol（循环左移位）指令和 ror（循环右移位）指令的操作数可以是寄存器或存储器中的字节、字或者双字。每一位从一端移出的位，被复制到目的操作数的另一端。另外，最后移出的位被复制到另一端，同时也被复制到进位标志位 CF，溢出标志位 OF 也受循环移位指令的影响。对于多位循环移位，OF 没有定义，因为与一位循环移位的定义类似，因此，本书没有对它进行定义。

例如，假定DX寄存器的数是D25E，执行指令

```
rol  dx,  1
```

用二进制表示，其运算为：

```
1101  0010  0101  1110
```

结果是10101 0100 1011 1101或A4BD。由于最左边的一位1循环移位到了右边，因此，进位标志CF被置为1。

循环移位指令的操作码和字节数与移位指令一样。具体的内容如图6-6所示。

循环移位指令

```
ror     eax, 4          ; shift next digit to right
```

可以用来替换过程hexToAscii（图6-7）中的移位指令。EAX的值最后保持不变，因为8次循环移位，每次移动4位，循环移位后，所有的位都回到原来的位置。在这个特定的程序中，循环移位没有什么优势，但它可用于类似的应用。

此外，还有一对循环移位指令rcl（带进位的循环左移位）和rcr（带进位的循环右移位）。这些指令都把进位标志位CF看成是目的操作数的一部分。这就是说，指令rcl eax, 1执行的结果是将EAX的0到30位左移一个位置，把原来第31位的值复制到CF，同时把原来CF的值复制给EAX的第0位。显然，带进位的循环移位指令影响CF，也会影响标志OF，但不影响其他的标志位。带进位循环移位指令的操作码与相应的移位指令一样，具体的内容如图6-6所示。

练习6.2

1. 对于下面的问题，假定给出指令执行前的值，求指令执行后的值。

指令执行之前	执行指令	指令执行之后
a. AX: A8 B5	shl ax, 1	AX, CF, OF
b. AX: A8 B5	shr ax, 1	AX, CF, OF
c. AX: A8 B5	sar ax, 1	AX, CF, OF
d. AX: A8 B5	rol ax, 1	AX, CF
e. AX: A8 B5	ror ax, 1	AX, CF
f. AX: A8 B5 CL: 04	sal ax, cl	AX, CF
g. AX: A8 B5	shr ax, 4	AX, CF
h. AX: A8 B5 CL: 04	sar ax, cl	AX, CF
i. AX: A8 B5 CL 04	rol ax, cl	AX, CF
j. AX: A8 B5	ror ax, 4	AX, CF
k. AX: A8 B5 CF: 1	rcl ax, 1	AX, CF
l. AX: A8 B5 CF: 0	rcr ax, 1	AX, CF

2. 计算寄存器 EAX 中的无符号整数除以 32，比较不同方法所用的时钟周期数和目标代码字节数。

```
a. mov  edx,0     ; extend value to doubleword
   mov  ebx,32    ; divisor
   div  ebx       ; value div 32
b. shr  eax,1     ; divide by 2
   shr  eax,1     ; divide by 2
   shr  eax,1     ; divide by 2
   shr  eax,1     ; divide by 2
   shr  eax,1     ; divide by 2
c. shr  eax,5     ; divide by 32
```

3. 计算寄存器 EAX 中的值乘以 32，比较不同方法所用的目标代码字节数。

```
a. mov  ebx,32    ; multiplier
   mul  ebx       ; value * 32
b. imul eax,32    ; value * 32
c. shl  eax,1     ; multiply by 2
   shl  eax,1     ; multiply by 2
   shl  eax,1     ; multiply by 2
   shl  eax,1     ; multiply by 2
   shl  eax,1     ; multiply by 2
d. shl  eax,5     ; multiply by 32
```

4. 假设 value1、value2 和 value3 在存储器中各占一个字节，每个字节存储一个无符号整数。假设第一个值不大于 31，这样它最多有 5 个有效位，开头最少有三个 0。同样，假设第二个值不大于 15（4 个有效位），第三个值不大于 127（7 个有效位）。
   a. 写出代码，把这三个数压缩为一个 16 位的字存放在寄存器 AX 中，把 value1 的低 5 位按顺序存入 AX 的 11 ～ 15 位，value2 的低 4 位按顺序存入 AX 的 7 ～ 10 位，value3 的低 7 位存入 AX 的 0 ～ 6 位。
   b. 写出代码，把寄存器 AX 中的这个 16 位数分解为 5 位、4 位和 7 位的三个数，每个数的左边空余位填 0，以补足 8 位，结果分别存放到 value1、value2 和 value3 中。
5. 指令

```
mov   ebx, eax     ; value
shl   eax, 1       ; 2*value
add   eax, ebx     ; 3*value
```

是将 EAX 中的值乘以 3。使用移位和加法指令，编写和上述指令序列类似的代码，计算 EAX 中的值乘以 5、7、9 和 10。

编程练习 6.2

1. 编写一个 NEAR32 的过程 binaryToAscii，它能把一个双字转换成 32 位的字符串，字符 0 和 1 表示它的值是一个无符号二进制数。该过程有两个参数，通过堆栈传递。

a. 双字值；

b. 32 位长的目的字符串地址。

该过程将从栈中取出参数，并且不能修改任何寄存器的内容。每次使用循环移位指令从左到右提取一位数字，使用 jc 或 jnc 指令来查看进位标志位。

2. 一个字节可以采用 3 个八进制数表示。第 7 和第 6 位决定左边的一个八进制数（不会超过 4）；第 5、4 和 3 位决定中间的一个八进制数；第 2、1 和 0 位决定右边的一个八进制数。例如，11010110_2 是 $11\ 010\ 110_2$ 或 326_8。一个字用八进制数表示的方法是用 2-3-3 模式，分别表示高 8 位和低 8 位字节。编写一个 NEAR32 过程，一个字的值可以采用“分离的八进制”表示，它是通过对高位和低位字节分别应用 2-3-3 模式来实现。写一个 NEAR32 的过程 splitOctal，它能实现把一个字转换成 6 字符的字符串，该字符串是用分离八进制表示的数值。这个过程有两个参数，通过堆栈传递。

a. 字的值；

b. 6 位长的目的字符串地址。

该过程将从栈中取出参数，并且不能修改任何寄存器的内容。

6.3 本章小结

本章探讨了对一个字节、字、或双字操作数中的位进行运算的各种 80x86 指令。逻辑指令 and、or 和 xor 可对源操作数和目的操作数的对应位进行布尔运算。这些指令的功能包括对目的操作数所选定的位进行置 1 或清 0。not 指令是对目的操作数中每一位取反，把每位由 1 变为 0，由 0 变为 1。test 指令和 and 指令相同，但 test 指令只影响标志位，目的操作数不改变。

移位指令把目的操作数的位左移或右移。移位指令分为一位移位指令和多位移位指令。一位移位指令中，第二个操作数是 1；多位移位指令中，用 CL 或立即数作为第二个操作数，并且指定目的操作数移动的位数。除了算术右移负数填 1 外，其余所有的一位移位运算的空出位都填 0。如果用移位指令来计算乘以或除以 2、4、8 或 2 的更高次幂，则是一种更有效、方便的方法。

循环移位指令同移位指令相似。但是，循环移位指令中从目的操作数一端移出的位填补到另一端的空位。移位或循环移位指令可以和逻辑指令一起用于取出一个存储单元中的一组位，或把多个值压缩为一个字节或字。

第7章 浮点运算

本书集中了以二进制补码为主的整型数的多种表示方法，因为所有 80x86 微处理器都有一些处理二进制补码的指令。大多数 80x86 微处理器系统也具有操纵以浮点数形式存储的数据的指令。

7.1 节讨论了存储浮点数的格式。7.2 节描述了 80x86 的浮点数的结构，它有一套全新的寄存器和指令。7.3 节论述了如何实现浮点数和其他可读格式表示之间的相互转化。7.4 节简要介绍了如何在 C++ 代码中嵌入汇编语言，其中用 C++ 实现输入输出操作，用汇编语言来实现浮点数运算。

7.1 浮点数表示法

浮点数是以近似科学记数法的形式来存储数字。首先用一个例子来说明如何把十进制数 78.375 转换成 32 位的 IEEE 单精度格式。浮点表示法由 IEEE 计算机协会标准委员会提出，得到 IEEE 标准委员会和美国国家标准协会（ANSI）的认可，它是 Intel 80x86 处理器用到的一种浮点格式。首先，78.375 必须转换成二进制。整数部分的转换很简单，十进制的 78 所对应的二进制是 1001110。在二进制中，小数点右边部分（二进制数的“.”不能确切地说是十进制的“.”），分别对应的是 2 的负幂（1/2，1/4，1/8，等等），正如十进制数中对应 10 的负幂一样（1/10，1/100，1/1000，等等）。因为 $0.375=3/8=1/4+1/8=.01_2+.001_2$，从而有 $0.375_{10} = 0.011_2$。把整数部分和小数部分组合在一起，因此，$78.375_{10}=1001110.011_2$。

然后，用小数点前面是 1 的数作为尾数，二进制科学记数法表示为：

$1001110.011_2=1.001110011\times 2^6$

二进制的科学记数法的指数与十进制记数一样精确，将二进制的“.”从右向左移动产生尾数来记数。这里的表示法是混合使用的，写成 2^6 比写成 10^{110} 更恰当些，但用十进制表示更方便。最后把它们合在一起就是该数的浮点表示。

- 左边的 0 位表示正数（1 表示负数）。
- 1000 0101 表示指数。即指数 6，加上**偏值** 127，和为八位数的 133。
- 00111001100000000000000，小数部分的开始位是 1，用 0 移走或填满右边的部分以达到 23 位。

整个数表示为 0 10000101 00111001100000000000000。重新分组可得到 0100 0010 1001 1100 1100 0000 0000 0000 或十六进制的 429CC000。

这个例子容易计算出来，因为十进制数 78.375 的小数部分 0.375 可由 2 的负幂求和得到。大多数的数字并不是这么容易表示，通常是用一个二进制小数来近似表示一个十进制数的小数部分，一种选择近似值的算法将在练习中给出。

总之，通过下面步骤可以把十进制数转换成 IEEE 的单精度格式：

1）浮点数的最高位如果为 0 则表示正数，1 表示负数。

2）用二进制来表示无符号数。

3）用二进制科学记数法表示二进制数，如 $f_{23}f_{22}\cdots f_0\times 2^e$，其中，$f_{23}=1$。包括尾部的 0 总共有 24 位小数。

4）将指数 e 加上偏值 127_{10}，得到的和用二进制表示，符号位后的 8 位就是所要的结果（加偏值 127_{10} 是将指数转换成有符号数）。

5）$f_{22}\ f_{21}\cdots f_0$ 形式的小数部分是浮点数的最后 23 位。最高位 f_{23}（通常是 1）可以省略掉。

在汇编语言程序的数据段，指示性语句 REAL4 会产生 4 字节的存储空间，可用来初始化一个单精度浮点值。例如：

```
number1   REAL4    78.375
```

在列表文件中所产生的结果是：

```
00000000  429CC000      number1     REAL4     78.375
```

80x86 所使用的第二种浮点格式是 64 位的 IEEE 双精度格式。它与单精度格式很类似，只不过指数是以 1023 作为偏值，且使用 11 位来存储，小数部分需要 52 位来存储。上例的负数形式为：

$-78.375_{10} = -1.001110011\times 2^6$

符号位是 1，指数为 6 + 1023 = 1029 = 100 0000 0101，小数部分为 00111001100…0 共 52 位，不足部分尾部用 0 填充。这些内容组合成一个四字长的数，并重新分组为 1100 0000 0101 0011 1001 1000 0000…0000，或者是十六进制数 C053980000000000。

在汇编语言程序的数据段，指示性语句 REAL8 可产生 8 字节的存储空间，可用来初始化一个双精度浮点值。例如：

```
number2   REAL8    -78.375
```

在列表文件所产生的结果是：

```
00000004                number2    REAL8    -78.375
        C053980000000000
```

因为该语句较长，所以它以两行的方式显示。

80x86 所使用的第三种浮点格式是 80 位的**扩展实数**。它也是以符号位开始，且使用偏值为 16 383 的 15 位指数，小数部分是 64 位长。与单精度格式和双精度格式不同的是：它的起始位为 1。再次使用 78.375 作为例子

$78.375_{10} = 1.001110011\times 2^6$

符号位是 0，指数是 6 + 16 383 = 16 389 = 100 0000 0000 0101，小数部分为 100111001100…0 共 64 位长，不足部分尾部用 0 填充。这些内容组合在一起可形成一个 10 字节数，重新分组为 0100 0000 0000 0101 1001 1100 1100 0000…0000，或者是十六进制数 40059CC 0000000000000。

在汇编语言程序的数据段，指示性语句 REAL10 可产生 10 字节的存储空间，可用来初始化

一个扩展实数的浮点数。例如：

```
number3   REAL10  78.375
```

在列表文件所产生的结果是：

```
0000000C                    number3 REAL10 78.375
        40059CC0000000000000
```

因为该语句较长，所以它以两行的方式显示。

在指数的表示中使用偏值来存储二进制补码的有符号数也是一种可选方法。使用单精度浮点数格式，指数的最大偏值是 FF（8 位，每位都是 1），且 $127_{10} = 7F_{16}$，因此指数的最大值是 $FF-7F = 80_{16} = 128_{10}$。最大的小数部分是 1.11111111111111111111111，它近似于 $10.0_2 = 2_{10}$。这说明最大的单精度浮点数近似于 2^{128} 或 3.40×10^{38}。其他形式最大值与最小值的计算留作练习。

单精度有 24 位小数部分，即开始位 1，加上存储的 23 位。因为 $2^{24} = 16\ 777\ 216$，所以，单精度格式大概可精确地表示 7 位十进制数。其他精度的计算留作练习。图 7-1 总结了三种浮点格式的表示法。

格式	总位数	指数位数	小数位数	近似最大值	近似最小值	近似十进制精度
单精度	32	8	23	3.40×10^{38}	1.18×10^{-38}	7 位
多精度	64	11	52	1.79×10^{308}	2.23×10^{-308}	15 位
扩展实数	80	15	64	1.18×10^{4932}	3.37×10^{-4932}	19 位

图 7-1　浮点格式

上面描述的格式都是**正规**的表示法，即二进制科学记数法的尾数是以 1 和小数点开始。这是最常用的表示法。显然，0 不能正规化表示。所有位都是 0 的位模式表示的是 +0。除了第 1 位是 1，后面的位都是 0 表示 −0。IEEE 标准和 Intel 结构还提供了其他非正规化数字的表示法。IEEE 标准甚至定义了 +∞、−∞和 NaN（非数字）的表示法，NaN 格式在计算结果不可用数字表示时使用。

显然，并不是所有的实数都能用给定的浮点体系表示，实数是无穷的，而能用 32、64、80 位的位模式表示的数是有限的。许多十进制数，如例子中的 73.375，以及其他的数字都能正确地表示，除非数值太大。可以正确表示的数字并不是均匀分布的。在实数线上，0.0 附近的数非常密，且随着数字的变大会越来越稀疏。也就是说，对于无法准确表示的实数，如果它非常接近 0.0，那么浮点的近似值将非常接近。但是如果它很大，那么该数与其最佳的浮点近似值的差别可能会相差很大。

练习 7.1

使用单精度浮点数来表示下面的每一个数。

1. 175.5　　2. −1.25

3. −11.75　　4. 45.5
5. 0.09375　　6. −0.0078125
7. 3160.0　　8. −31.0

使用双精度浮点数来表示下面的每一个数。

9. 175.5　　10. −1.25
11. −11.75　　12. 45.5
13. 0.09375　　14. −0.0078125
15. 3160.0　　16. −31.0

使用扩展实数来表示下面的每一个数。

17. 175.5　　18. −1.25
19. −11.75　　20. 45.5
21. 0.09375　　22. −0.0078125
23. 3160.0　　24. −31.0

计算下面每一个单精度浮点数所表示的十进制数。

25. C26A0000　　26. C26A0000

计算下面每一个双精度浮点数所表示的十进制数。

27. 407A44570A3D70A4　　28. BFA4000000000000

计算下面每一个扩展实数浮点数所表示的十进制数。

29. 4008FA00000000000000　　30. BFFEA000000000000000

31. 说明 1.18×10^{-38} 是普通的单精度浮点数所能表示的近似最小值。
32. 说明 1.79×10^{308} 是普通的双精度浮点数所能表示的近似最大值。
33. 说明 2.23×10^{-308} 是普通的双精度浮点数所能表示的近似最小值。
34. 说明 1.18×10^{4932} 是普通的扩展实数浮点数所能表示的近似最大值。
35. 说明 3.37×10^{-4932} 是普通的扩展实数浮点数所能表示的近似最小值。
36. 说明双精度浮点数能精确地表示大概 15 位的十进制。
37. 说明扩展实数浮点数能精确地表示大概 19 位的十进制。
38. 大多数十进制的小数部分不能够通过二进制来准确表示。假定 x 是 0 到 1 之间的十进制数。一种获得 n 位二进制的近似方法是重复地计算 2^{-j} 是否“满足”它在 x 中的值，j=1, 2, …, n。如果相等则相应的位就等于 1 并从 x 中减去 2^{-j}，否则相应的位就为 0。扩展这种思想来实现相应的伪代码算法。

7.2 80x86 浮点体系

Intel 80x86 微处理器的浮点运算器（Floating-Point Unit，FPU）几乎独立于其他芯片。它有自己的内部寄存器，完全独立于整数运算所使用的 80x86 寄存器。它通过指令的执行来完成浮点运算，包括普通的加法或乘法运算，以及如超越函数求值之类的复杂运算。它不仅可以在存储器之间传递浮点型操作数，还可以在协微处理器之间传递整数或者 BCD 操作数。当非浮点形式的数据存储到浮点寄存器时，它必须转换成浮点数。同样，当内部的浮点数存储到存储器时，

它必须转换成整数或 BCD 操作数。

FPU 有 8 个数据寄存器，每个寄存器是 80 位长。扩展实数的浮点格式可用来表示存储在这些寄存器中的数字。这些寄存器基本上是作为堆栈组织在一起的，例如：如果使用 fld（浮点载入）指令将一个数值从存储器传递到浮点型单元，那么该数值被放入堆栈顶端的寄存器，数据存储在栈顶，同时其他寄存器依次下推。然而，有些指令能访问栈顶之下的寄存器，因此该结构并不是一个纯粹的堆栈。

8 个浮点寄存器的名称分别为：

- ST，栈顶，也称为 ST(0)。
- ST(1)，仅居于栈顶之下的寄存器。
- ST(2)，居于 ST(1) 之下的寄存器。
- ST(3)，ST(4)，ST(5) 和 ST(6)，道理同上。
- ST(7)，是居于栈底的寄存器。

除了 8 个数据寄存器之外，浮点运算单元还有几个 16 位的控制寄存器。一些**状态字**的位可通过比较指令赋值，为了让基于浮点比较操作的 80x86 体系能执行条件转移指令，必须对这些位进行检查。FPU 的**控制字**必须设置，以确保某些位取整。

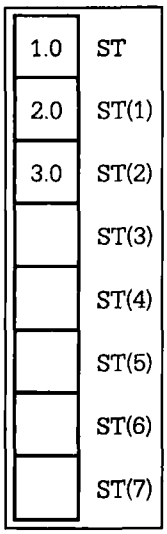
在考虑浮点指令之前，必须注意，每一个浮点助记符是由字母 F 开始，任何非浮点指令的首字母都不会使用该字母。大多数的浮点指令作用于栈顶 ST，其他操作数存放在另一个浮点型寄存器或存储器中。浮点型指令不能在一个通用寄存器（如 EAX）和一个浮点型寄存器之间传递数据，要实现这样的传递，必须使用一个存储器单元进行立即存储（但是，指令存储状态字或者控制字要传递给寄存器 AX）。

浮点指令必须按组检查，开始指令为操作数进栈。图 7-2 列出了这些助记符。本书没有考虑浮点指令的操作码。

助记符	操作数	作用
fld	存储器（实数）	存储器中的实数入栈
fild	存储器（整数）	存储器中的整数转成浮点数，并入栈
fbld	存储器（BCD）	存储器中的 BCD 转成浮点数，并入栈
fld	st(*num*)	浮点寄存器的内容入栈
fld1	（无）	1.0 入栈
fldz	（无）	0.0 入栈
fldpi	（无）	π(pi) 入栈
fldl2e	（无）	$\log_2(e)$ 入栈
fldl2t	（无）	$\log_2(10)$ 入栈
fldlg2	（无）	$\log_{10}(2)$ 入栈
fldln2	（无）	$\log_e(2)$ 入栈

图 7-2 浮点数载入指令

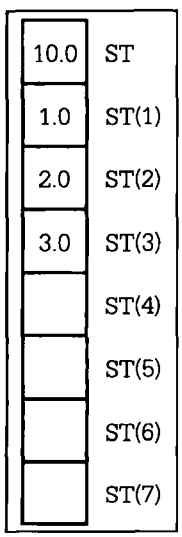
下面举例说明浮点指令是如何工作的。假定浮点指令寄存器的堆栈内容如下：



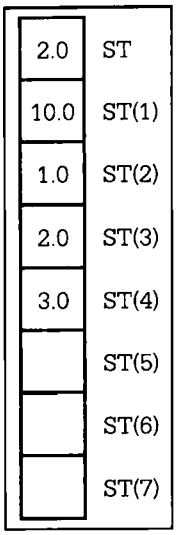
其中的数值用十进制表示，而不是 IEEE 浮点数格式。如果数据段包含：

```
fpValue   REAL4   10.0
intValue  DWORD   20
```

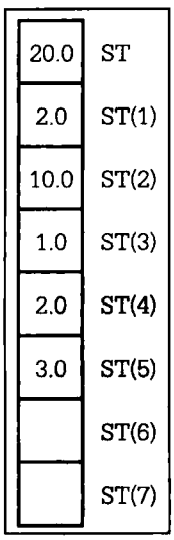
那么汇编后，fpValue 的值将是 41200000，且 intValue 的值为 00000014。如果执行指令 fld fpValue，寄存器堆栈将是：



堆栈的初始值都下移一个寄存器位置。基于这些值，如果执行指令 fld st(2)，寄存器堆栈将是：



值得一提的是，ST(2) 的值 2.0 已经到达栈顶，但是没有出栈。基于这些值，假定现在执行指令 `fild intValue`。寄存器堆栈的新内容将是：



这里 32 位的数 00000014 被转换成 80 位的扩展浮点型实数。一个整型操作数必须是单字长、双字长或四字长，字节长的整型操作数是不允许的。

最后，如果按顺序执行指令 fldz 和 fldpi，寄存器堆栈将包含：

堆栈的空间现在已满，没有值能再入栈，除非其他的值出栈，或堆栈被清空。指令 finit 能初始化浮点单元，并清空 8 个寄存器的内容。通常在代码开始部分，使用浮点单元的程序会包含下面的语句：

```
finit      ; initialize the math coprocessor
```

这条语句出现在代码的开始部分，程序中可能需要对浮点型单元重新初始化，但通常没有这个

必要，因为从堆栈中弹出的值不会继续堆积在堆栈中。

3.14	ST
0.0	ST(1)
20.0	ST(2)
2.0	ST(3)
10.0	ST(4)
1.0	ST(5)
2.0	ST(6)
3.0	ST(7)

使用工具 Windbg 可跟踪浮点数的运算。图 7-3 中，屏幕左边的面板显示下面代码执行的过程，右边的面板显示对应的浮点窗口。数据段包含：

```
two        DWORD  2
three      DWORD  3
fpValue    REAL4  10.0
intValue   DWORD  20
```

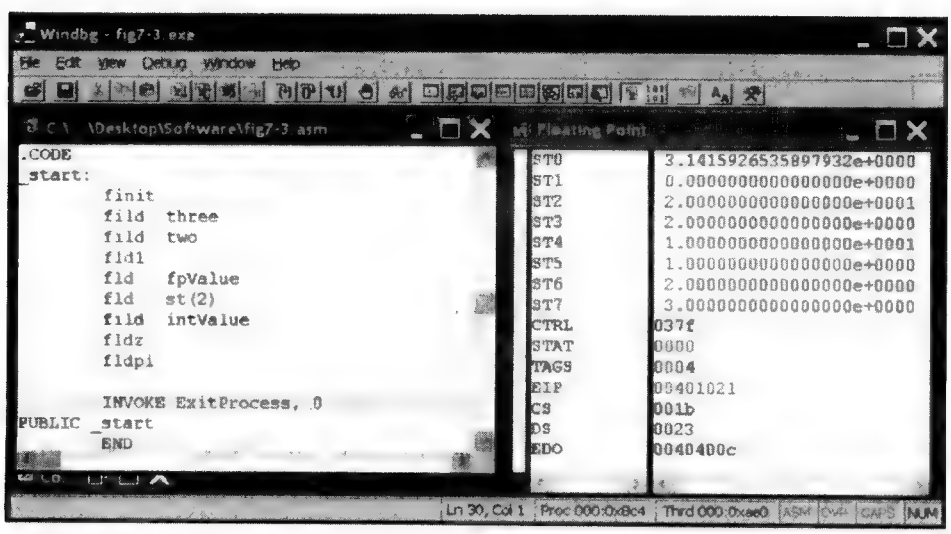


图 7-3 Windbg 窗口显示的浮点数指令的执行

图 7-4 列出了一些浮点型指令，它们可用于将栈顶数据复制到存储器或其他浮点型寄存器。这些指令大多都是成对的：每对指令中的一条仅仅是复制 ST 到目的地，另一条指令除复制 ST

到它的目的地外，它还将 ST 从寄存器堆栈中取出来。

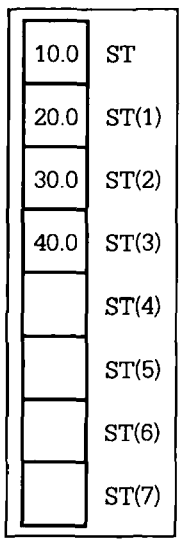
助记符	操作数	作用
fst	st(*num*)	复制 ST 的值替换 ST(*num*) 的内容，只有 ST(*num*) 的值会改变
fstp	st(*num*)	复制 ST 的值替换 ST(*num*) 的内容，ST 的值出栈
fst	存储器（实数）	复制 ST 的值存储为存储器的实数，栈不受影响
fstp	存储器（实数）	复制 ST 的值存储为存储器的实数，ST 的值出栈
fist	存储器（整数）	复制 ST 并转换成整数，存入存储器
fistp	存储器（整数）	复制 ST 并转换成整数，存入存储器，ST 的值出栈
fbstp	存储器（BCD）	复制 ST 并转换成 BCD，存入存储器，ST 的值出栈

图 7-4 浮点数数据存储指令

举例说明这些指令的不同作用。假定指令

```
intValue  DWORD  ?
```

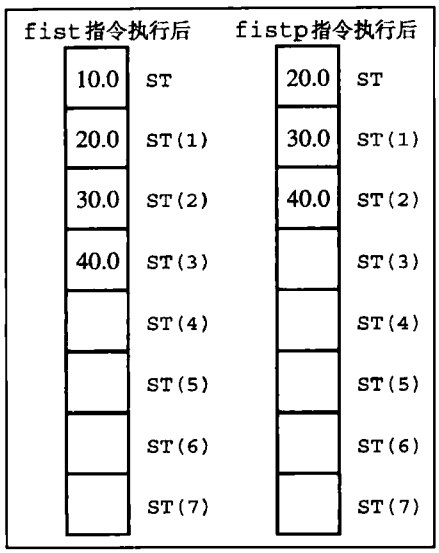
在数据段进行编码且浮点寄存器堆栈的内容为：

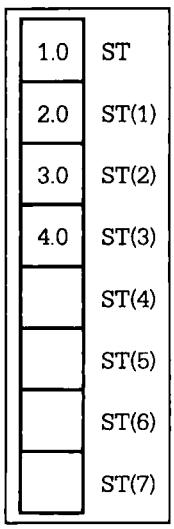


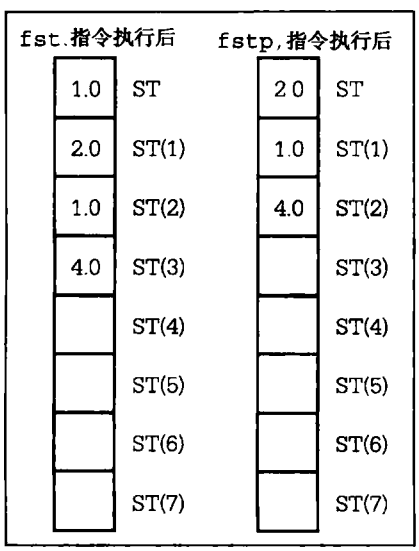
下图左显示了指令 fist intValue 执行后的堆栈情况，下图右显示了指令 fistp intValue 执行后的堆栈情况。在这两种情况下，intValue 中的值都是 0000000A，这是浮点数 10.0 的双字长二进制补码的整型表示。

当目的操作数是浮点寄存器时，情况就稍微复杂一些。假定在执行时，浮点寄存器堆栈的内容为：

下图左显示了执行 fst st(2) 后的堆栈内容，下图右显示了执行 fstp st(2) 后的堆栈情况。在第一种情况下，复制的 ST 已经保存在 ST(2) 中，在第二种情况下，复制 ST 后，然后 ST 的内容被弹出。


fist指令执行后
10.0 ST
20.0 ST(1)
30.0 ST(2)
40.0 ST(3)
ST(4)
ST(5)
ST(6)
ST(7)
fistp指令执行后
20.0 ST
30.0 ST(1)
40.0 ST(2)
ST(3)
ST(4)
ST(5)
ST(6)
ST(7)


1.0 ST
2.0 ST(1)
3.0 ST(2)
4.0 ST(3)
ST(4)
ST(5)
ST(6)
ST(7)


fst.指令执行后
1.0 ST
2.0 ST(1)
1.0 ST(2)
4.0 ST(3)
ST(4)
ST(5)
ST(6)
ST(7)
fstp,指令执行后
2 0 ST
1.0 ST(1)
4.0 ST(2)
ST(3)
ST(4)
ST(5)
ST(6)
ST(7)

除了上面所列出的载入和存储指令之外，浮点运算器还有一条 fxch 指令，用于与其他的浮点寄存器交换 ST 的内容。如果没有操作数，指令

```
fxch          ; exchange ST and ST(1)
```

将交换栈顶和 ST(1) 的内容。

如果有一个操作数，例如：

```
fxch  st(3)   ; exchange ST and ST(3)
```

将交换 ST 和特定寄存器的内容。

图 7-5 对浮点加法指令进行说明。这里有多种加法形式：将 ST 中的内容加到其他寄存器中，将任一寄存器中的内容加到 ST 中，将存储器中的一个实数加到 ST 中，或者将存储器中的一个整型数加到 ST 中。在将栈顶内容加到另一个寄存器中后，指令 faddp 将它从栈顶取出，这样两个操作数都改变了。

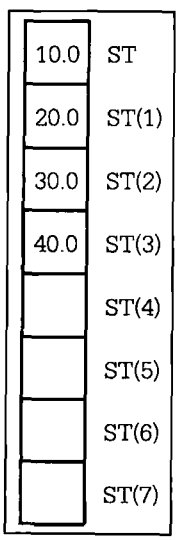
助记符	操作数	作用
fadd	（无）	ST 和 ST(1) 出栈，计算它们的和，并入栈
fadd	st(*num*)，st	计算 ST(*num*) 和 ST 的和，并用和替换 ST(*num*)
fadd	st,st(*num*)	计算 ST 和 ST(*num*) 的和，并用和替换 ST
fadd	存储器（实数）	计算 ST 和存储器中实数的和，并用和替换 ST
fiadd	存储器（整数）	计算 ST 和存储器中整数的和，并用和替换 ST
faddp	st(*num*)，st	计算 ST(*num*) 和 ST 的和，并用和替换 ST(*num*)，ST 出栈

图 7-5 浮点型加法指令

举例说明浮点加法指令的工作过程。假定数据段包含下面的语句：

```
fpValue   REAL4  5.0
intValue  DWORD  1
```

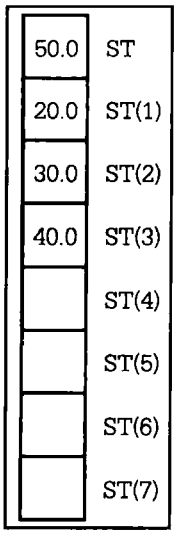
且浮点寄存器堆栈包含：



指令

```
fadd st,st(3)
```

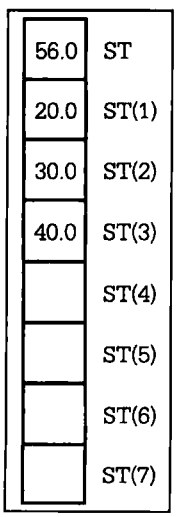
执行之后，堆栈的内容为：



在此基础上，执行下面两条指令

```
fadd  fpValue
fiadd  intValue
```

执行之后，堆栈的内容是：



最后，如果指令

```
faddp   st (2) , st
```

被执行，堆栈的内容将是：

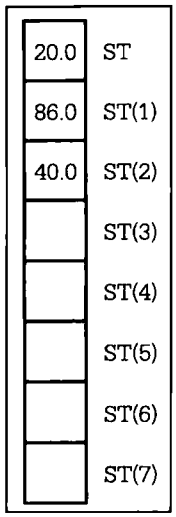
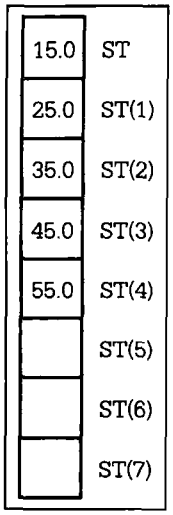


图 7-6 对浮点减法指令进行说明。前 6 个指令与相应的加法指令很类似。接下来的 6 个指令与加法指令也是一样的，除了操作数是以相反的顺序减去。这种方法很容易实现，因为减法运算是不可交换的。

助记符	操作数	作用
fsub	（无）	ST 和 ST(1) 出栈，计算 ST(1)−ST 的差并入栈
fsub	st(*num*)，st	计算 ST(*num*)−ST 的差，并用差替换 ST(*num*)
fsub	st,st(*num*)	计算 ST−ST(*num*) 的差，并用差替换 ST
fsub	存储器（实数）	计算 ST− 存储器中实数的差，并用差替换 ST
fisub	存储器（整数）	计算 ST− 存储器中整数的差，并用差替换 ST
fsubp	st(*num*),st	计算 ST(*num*)−ST 的差，并用差替换 ST(*num*)，ST 出栈
fsubr	（无）	ST 和 ST(1) 出栈，计算 ST−ST(1) 的差并入栈
fsubr	st(*num*),st	计算 ST−ST(*num*) 的差，并用差替换 ST(*num*)
fsubr	st,st(*num*)	计算 ST(*num*)−ST 的差，并用差替换 ST
fsubr	存储器（实数）	计算存储器中实数 −ST 的差，并用差替换 ST
fisubr	存储器（整数）	计算存储器中整数 −ST 的差，并用差替换 ST
fsubpr	st(*num*),st	计算 ST−ST(*num*) 的差，并用差替换 ST(*num*)，ST 出栈

图 7-6 浮点型减法指令

下面举例来说明这些减法指令功能上的不同，假定浮点寄存器堆栈内容为：



下面的两个图是执行指令 `fsub st,st(3)` 和 `fsubr st,st(3)` 之后的结果。

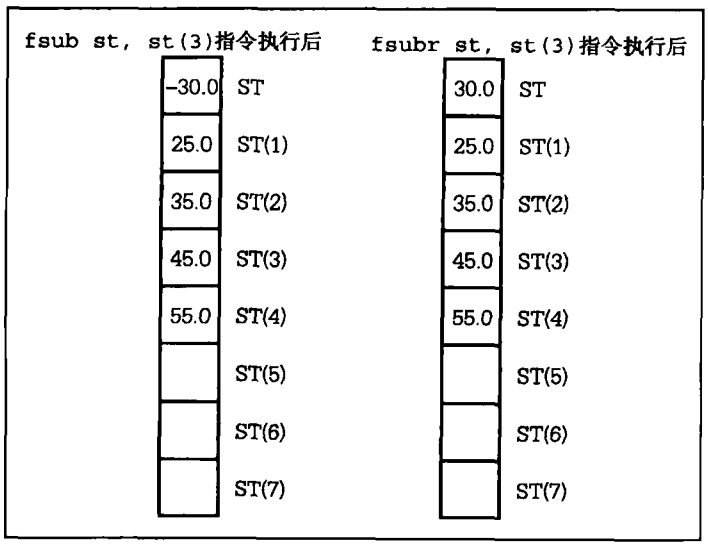


图 7-7 和图 7-8 分别列出了乘法和除法指令。乘法指令与图 7-5 中的加法指令具有相同的格

助记符	操作数	作用
fmul	（无）	ST 和 ST(1) 出栈，计算它们的乘积并入栈
fmul	st(*num*),st	计算 ST(*num*) 和 ST 的乘积，并用乘积替换 ST(*num*)
fmul	st,st(*num*)	计算 ST 和 ST(*num*) 的乘积，并用乘积替换 ST
fmul	存储器（实数）	计算 ST 和存储器中实数的乘积，并用乘积替换 ST
fimul	存储器（整数）	计算 ST 和存储器中整数的乘积，并用乘积替换 ST
fmulp	st(*num*),st	计算 ST(*num*) 和 ST 的乘积，并用乘积替换 ST(*num*)，ST 出栈

图 7-7 浮点型乘法指令

式。除法指令与图 7-6 中的减法指令具有相同的格式。

助记符	操作数	作用
fdiv	（无）	ST 和 ST(1) 出栈，计算 ST(1)/ST 的商并入栈
fdiv	st(*num*),st	计算 ST(*num*)/ST 的商，并用商替换 ST(*num*)
fdiv	st,st(*num*)	计算 ST/ST(*num*) 的商，并用商替换 ST
fdiv	存储器（实数）	计算 ST/ 存储器中实数的商，并用商替换 ST
fidiv	存储器（整数）	计算 ST/ 存储器中整数的商，并用商替换 ST
fdivp	st(*num*),st	计算 ST(*num*)/ST 的商，并用商替换 ST(*num*)，ST 出栈
fdivr	（无）	ST 和 ST(1) 出栈，计算 ST/ST(1) 的商并入栈
fdivr	st(*num*),st	计算 ST/ST(*num*) 的商，并用商替换 ST(*num*)
fdivr	st,st(*num*)	计算 ST(*num*) /ST 的商，并用商替换 ST
fdivr	存储器（实数）	计算存储器中实数 /ST 的商，并用商替换 ST
fidivr	存储器（整数）	计算存储器中整数 /ST 的商，并用商替换 ST
fdivpr	st(*num*),st	计算 ST/ST(*num*) 的商，并用商替换 ST(*num*)，ST 出栈

图 7-8　浮点型除法指令

图 7-9 描述了 4 条附加的浮点型指令，用于计算三角函数、指数函数和对数函数，本书没有涉及这些内容。

助记符	操作数	作用
fabs	（无）	ST:=|ST|（绝对值）
fchs	（无）	ST:=−|ST|（相反数）
frndint	（无）	对 ST 取整
fsqrt	（无）	用 ST 的平方根替换 ST 的内容

图 7-9　附加的浮点型指令

浮点运算器提供比较栈顶元素 ST 与第二操作数的指令集合，这些指令在图 7-10 中列出，回想一下，浮点运算器有一个称为状态字的 16 位控制寄存器。比较指令可以为状态字的第 14、10 和 8 位赋值；这些“条件码”的位分别称为 C3、C2 和 C0。它们的设置是根据下面的条件进行设定：

比较结果	C3	C2	C0
ST> 操作数	0	0	0
ST< 操作数	0	0	1
ST= 操作数	1	0	0

另外还有一种可能是两个操作数不可比。如果其中一个操作数是 IEEE 表示的无穷大或者不是一个数值（NaN）时，那么，这种情况就会出现。此时，这三位“条件码”位都设置为“1”。

助记符	操作数	功能
fcom	（无）	比较 ST 和 ST(1)
fcom	st(*num*)	比较 ST 和 ST(*num*)
fcom	存储器（实数）	比较 ST 和存储器中的实型数
ficom	存储器（整数）	比较 ST 和存储器中的整型数
ftst	（无）	比较 ST 和 0.0
fcomp	（无）	比较 ST 和 ST(1)，然后出栈
fcomp	st(*num*)	比较 ST 和 ST(*num*)，然后出栈
fcomp	存储器（实数）	比较 ST 和 存储器中的实型数，然后出栈
ficomp	存储器（整数）	比较 ST 和 存储器中的整型数，然后出栈
fcompp	（无）	比较 ST 和 ST(1)，然后出栈两次

图 7-10 浮点比较指令

如果比较操作的目的是为了决定程序的流程，简单地设置状态字中的标志位是没有用的。在 80x86 中，条件转移指令是参考标志寄存器的某些位，而不是浮点运算器中的状态字。因此，在用 80x86 指令（如 test 指令）测试状态字的一些位之前，状态字必须复制到存储器或者 AX 寄存器中。浮点运算器有两条指令可以用来存储状态字，图 7-11 对它们进行了总结，并列出了存储和设置控制字的指令。

80x86 浮点型和整型运算器能够并发执行指令，因此，在某些情况下，用汇编语言编程要特别小心，本书不对此展开讨论。

助记符	操作数	作用
fstsw	存储器字	复制状态寄存器给存储器字
fstsw	AX	复制状态寄存器给 AX
fstcw	存储器字	复制控制字寄存器给存储器字
fldcw	存储器字	复制存储器字给控制字寄存器

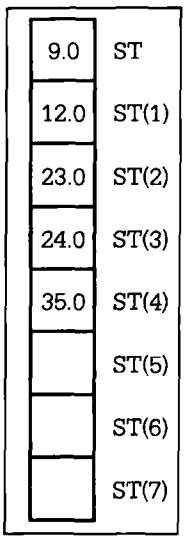
图 7-11 FPU 状态字和控制字的访问

练习 7.2

1. 假设一个程序的数据段内容如下：

```
fpValue     REAL4    0.5
intValue    DWORD    6
```

相关代码还没有执行，程序还没有修改它们的数值。浮点型寄存器堆栈内容如下：



假设这些数值在下列各指令执行之前都是正确的。而且下列指令都是独立执行的，不存在哪条指令先执行，哪条指令后执行的问题。请给出执行下列指令后，fpValue 和 intValue 的浮点型寄存器堆栈的内容。

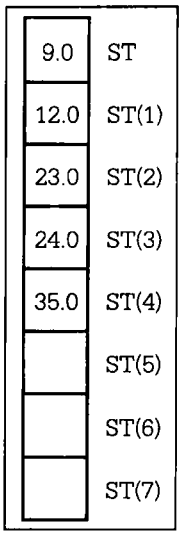
```
a. fld    st(2)          b. fld    fpValue
c. fild   intValue       d. fldpi
e. fst    st(4)          f. fstp   st(4)
g. fst    fpValue        h. fistp  intValue
i. fxch   st(3)          j. fadd
k. fadd   st(3), st      l. fadd   st, st(3)
m. faddp  st(3),st       n. fsub   fpValue
o. fisub  intValue       p. fisubr intValue
q. fsubp  st(3), st      r. fmul   st, st(4)
s. fmul                  t. fmul   fpValue
u. fdiv                  v. fdivr
w. fidiv  intValue       x. fdivp  st(2), st
y. fchs                  z. fsqrt
```

2. 假设某程序数据段内容如下：

```
fpValue    REAL4   1.5
intValue   DWORD   9
```

相关代码还未执行，程序还没有修改它们的数值。假设浮点型寄存器堆栈内容如下：

假设这些数值在下列各指令执行之前都是正确的。给出下列指令执行后状态字标志位 C3、C2 和 C0 的内容。



a. `fcom`　　b. `fcom st(3)`

c. `fcom fpValue`　　d. `ficom intValue`

对下面的两个指令，也给出执行之后堆栈的内容。

e. `fcomp`　　f. `fcompp`

7.3 浮点型指令编程

本节给出三个用浮点指令编程的例子。第一个例子程序是计算两个数平方和的平方根。第二个例子是实现 ASCII 码格式的十进制数到浮点数格式转换的过程，而第三个例子是实现浮点数到 ASCII 码转换的过程。

图 7-12 列出了第一个例子。value1 和 value2 是浮点型数值。`finit` 指令确保 FPU 寄存器的内容能被清空。第二条指令实现从存储器复制 value1 值给 ST，第三条指令是把 value 从 ST 复制给 ST，并把第一个堆栈的值下移到 ST(1)。第四条指令是在 ST 中计算 value1*value1 的值，同时 ST(1) 的值 “清空”（当然，每个浮点寄存器总是会有一些值）。对 value2 重复执行这三条指令。相加这两个数的平方和并计算平方根。

图 7-13 显示平方根存储和出栈前的状态。浮点窗口显示的是十进制记数法所表示 SP 的结果。通过手工计算可知，正确的答案是 1.3，但是，计算所得到的结果与正确的答案有一些差别。如果跟踪程序的运行，就会发现计算结果 1.2 是不准确的。

奇怪的是，最后 ST(6) 和 ST(7) 的内容是非 0 的。这是由寄存器堆栈的工作方式决定的。值的出栈是通过移动指针指向寄存器 ST 而不是实际地移动寄存器之间的位。图中右边浮点所显示的 ST(6) 和 ST(7) 的值表示逻辑上已出栈。80x86 的浮点体系并不会使用这些值。例如，在 `fstp` 指令之前使用 `fld st(7)` 将会产生一个错误情况。

第二个例子是实现简单的“ASCII 码到浮点数”转换的算法。图 7-14 给出该算法的代码，它与编程练习 6.1 所描述的 atoi 过程很类似。该程序通过参数扫描所给地址的存储器，并把字

```
; find the square root of the sum of the squares of two floating point numbers
; Author:  R. Detmer
; Date:    revised 8/2005

.386
.MODEL FLAT
ExitProcess PROTO NEAR32 stdcall, dwExitCode:DWORD

.STACK  4096                ; reserve 4096-byte stack

.DATA                       ; reserve storage for data
value1  REAL4   0.5
value2  REAL4   1.2
sqrt    REAL4   ?

.CODE
_start:
        finit               ; initialize floating point unit
        fld     value1      ; value1 in ST
        fld     st          ; value1 in ST and ST(1)
        fmul                ; value1*value1 in ST
        fld     value2      ; value2 in ST (value1*value1 in ST(1)
        fld     st          ; value2 in ST and ST(1)
        fmul                ; value2*value2 in ST
        fadd                ; sum of squares in ST
        fsqrt               ; square root of sum of squares in ST
        fstp    sqrt        ; store result, clearing stack

        INVOKE  ExitProcess, 0
PUBLIC _start
END
```

图 7-12　计算平方和的平方根程序

符转换成相应的浮点数。该程序可查找最高位的负号和十进制的小数点。如果遇到非数字字符，则扫描中断。

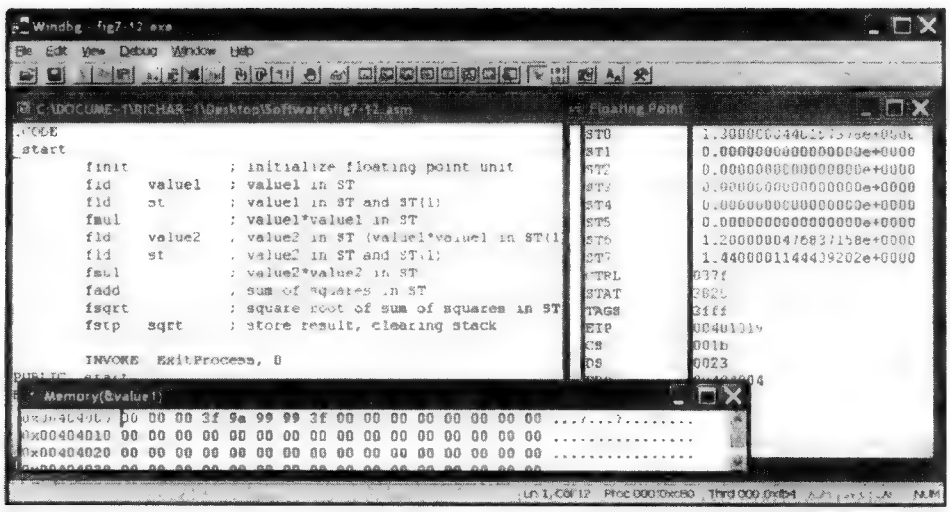


图 7-13　运行计算平方和的平方根程序

```
value := 0.0;
divisor := 1.0;
point := false;
minus := false;

point at first character of source string;
if source character = '-'
then
    minus := true;
    point at next character of source string;
end if;

while (source character is a digit or a decimal point) loop
    if source character = '.'
    then
        point := true;
    else
        convert ASCII digit to 2's complement digit;
        value := 10*value + float(digit);
        if point
        then
            multiply divisor by 10;
        end if;
    end if;
    point at next character of source string;
end while;

value := value/divisor;

if minus
then
    value := - value;
end if;
```

图 7-14 ASCII 码到浮点数的转换算法

算法用一个 NEAR32 过程来实现。该过程有一个参数，即字符串的地址，返回浮点值放在 ST 中。没有设置标志位来指出非法的情况，比如是否有多个负号或十进制小数点或非数字字符等等。图 7-15 是该过程的代码。

ASCII 码到浮点数转换算法的实现如图 7-15 所示。其中 value 使用 ST，divisor 使用 ST(1)。但是在一个短数据段中，为了修改 divisor，divisor 使用了 ST，而 value 使用了 ST (1)。输入代码后，指令

```
; ASCII-to-floating-point code
; author:  R. Detmer
; revised:  8/2005
.386
.MODEL FLAT
ExitProcess PROTO NEAR32 stdcall, dwExitCode:DWORD
.STACK  4096              ; reserve 4096-byte stack
.DATA
source  BYTE   "-78.375", 0
result  REAL4  ?

.CODE
_start:
        lea  eax, source                ; address parameter
        push eax                        ; push parameter
        call atof                       ; atof(source)
        fstp result                     ; get result from FPU
        INVOKE ExitProcess, 0
PUBLIC _start

atof PROC NEAR32
; convert ASCII string to floating point number
; Parameter passed on the stack: address of ASCII source string
; After an optional leading minus sign, only digits 0-9 and a decimal
; point are accepted – the scan terminates with any other character.
; The floating point value is returned in ST.
; Other FP registers, integer registers, and flags are unchanged.
; Local variables are stored on the stack:
;   ten    [EBP-4]  – always 10.0 after initial code
;   point  [EBP-8]  – Boolean, -1 for true, 0 for false
;   minus  [EBP-12] – Boolean, -1 for true, 0 for false
;   digit  [EBP-16] – next digit as an integer
         push ebp                       ; establish stack frame
         mov  ebp, esp
         sub  esp, 16                   ; stack space for local variables
         push eax                       ; save registers
         push esi
         pushfd                         ; save flags

         mov  DWORD PTR [ebp-4], 10     ; ten := 10

         fld1                           ; divisor := 1.0
         fldz                           ; value := 0.0
         mov  DWORD PTR [ebp-8], 0      ; point := false
         mov  DWORD PTR [ebp-12], 0     ; minus := false
         mov  esi, [ebp+8]              ; address of first source character

         cmp  BYTE PTR [esi], '-'       ; leading minus sign?
         jne  endifMinus                ; skip if not
         mov  DWORD PTR [ebp-12], -1    ; minus := true
         inc  esi                       ; point at next source character
```

图 7-15 ASCII 码到浮点数的转换程序

```
endifMinus:

whileOK:  mov  al, [esi]                ; get next character
          cmp  al, '.'                  ; decimal point?
          jne  endifPoint               ; skip if not
          mov  DWORD PTR [ebp-8], -1    ; point := true
          jmp  nextChar
endifPoint:
          cmp  al, '0'                  ; character a digit?
          jl   endwhileOK               ; exit if lower than '0'
          cmp  al, '9'
          jg   endwhileOK               ; exit if higher than '9'
          and  eax, 0000000fh           ; convert ASCII to integer value
          mov  DWORD PTR [ebp-16], eax  ; put integer in memory
          fimul DWORD PTR [ebp-4]       ; value := value * 10
          fiadd DWORD PTR [ebp-16]      ; value := value + digit
          cmp  DWORD PTR [ebp-8], -1    ; already found a decimal point?
          jne  endifDec                 ; skip if not
          fxch                          ; put divisor in ST and value in ST(1)
          fimul DWORD PTR [ebp-4]       ; divisor := divisor * 10
          fxch                          ; value to ST; divisor back to ST(1)
endifDec:
nextChar: inc  esi                      ; point at next source character
          jmp  whileOK
endwhileOK:

          fdivr                         ; value := value / divisor
          cmp  DWORD PTR [ebp-12], -1   ; was there a minus sign?
          jne  endifNeg
          fchs                          ; value := -value
endifNeg:
          popfd                         ; restore flags
          pop  esi                      ; restore registers
          pop  eax
          mov  esp, ebp
          pop  ebp
          ret  4                        ; return, removing parameter
atof      ENDP
          END
```

图 7-15 （续）

```
fld1          ; divisor := 1.0
fldz          ; value := 0.0
```

是对这两个变量进行初始化。注意 divisor 的值 1.0 是在 ST(1) 中结束，因为它是指令 fldz 下推入栈的。

其他的局部变量存储在程序注释所表明的堆栈地址中。值得一提的是，该代码编码时不仅使用 [ebp-x]，而且也使用操作数的大小和类型。

设计

value := 10*value + float(digit);

是通过下面的代码来实现的：

```
fimul DWORD PTR [ebp-4]        ; value := value * 10
fiadd DWORD PTR [ebp-16]       ; value := value + digit
```

注意：对于乘法运算，浮点运算器会把整数 10 转换成浮点型数；对于加法运算，浮点运算器会把整型数转成浮点型数。

为了实现“divisor 乘 10”，被乘数必须放在 ST 中，指令

```
fxch                     ; put divisor in ST and value in ST(1)
fimul DWORD PTR [ebp-4]  ; divisor := divisor * 10
fxch                     ; value to ST; divisor back to ST(1)
```

进行 divisor 和 value 的交换，实现了乘法，将结果放入 ST，并重新交换回来。

接下来是实现“value :=value/divisor”的指令

```
fdivr          ; value := value / divisor
```

从 ST 中取出 value 值，并从 ST(1) 中获取 divisor，计算商并把它返回给 ST。注意：这里如果使用指令 fdiv 计算“divisor/value”，那么结果是不正确的。如果 ASCII 码字符串是以负号开始的话，指令 fchs 将改变 value 的符号。

图 7-15 的代码包含了 atof 过程的一个简单测试驱动程序。图 7-16 显示的是 atof 调用之后，Windbg 的执行状态，该状态正是结果从浮点堆栈中取出之前的状态。跟踪整个程序的运行，并

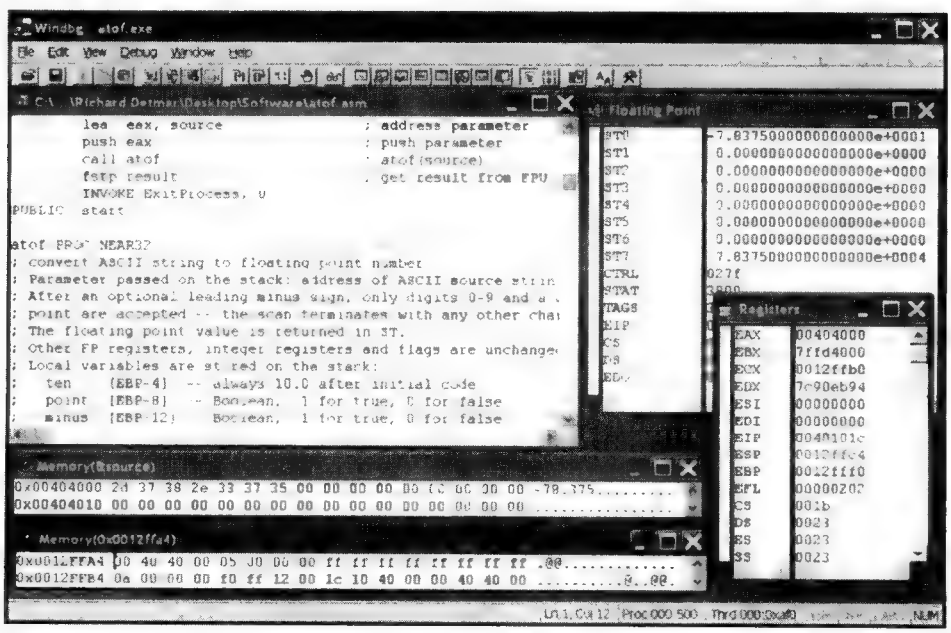


图 7-16 ASCII 码到浮点数转换程序的执行

观察 ST 中的累加值是很有意义的。

最后，考虑把单精度浮点数参数转换成“E 记数法”的过程。该过程生成一个 12 字节长的 ASCII 码字符串，包括

- 一个起始位的负号或空格
- 一个数字
- 一个十进制小数点
- 五个数字
- 字母 E
- 加号或减号
- 两个数字

这个字符串表示十进制数的科学记数法。例如，对于十进制小数 145.8798，该程序将它转换成字符串 b1.458880E+02。其中，b 代表空格。注意，这个 ASCII 码字符串有个四舍五入的数值。

图 7-17 给出了过程“浮点数到 ASCII 码转换”的程序代码。起始位的空格或负号产生后，在剩余字符真正转换前，还必须先取出这些字符。数值反复地乘以或除以 10，直到它的值大于等于 1.0 且小于 10.0 为止。如果初始值小于 1.0，那么必须使用乘法运算；乘法的次数是科学记数法中 10 的负次幂。对于初始值是等于或大于 10.0，那么必须使用除法运算，除法的次数是科学记数法中 10 的正次幂。

在小数点后面只显示五位数字。1.0 和 10.0 之间的值是通过加 0.000005 后四舍五入得到；如果小数点后的第 6 位是 5 或比 5 大，它就能反映出实际所表示的数。加法的结果可能等于或大于 10.0，如果这样，那么这个值将再除以 10，指数加 1。

对于大于等于 1.0 但小于 10.0 的数，在小数点前将其截断，只取一位整数。将这位整数和

```
point at first destination byte;

if value ≥ 0
then
    put blank in destination string;
else
    put minus in destination string;
    value := -value;
end if;
point at next destination byte;

exponent := 0;
if value ≠ 0
then
    if value > 10
    then
        until value < 10 loop
            divide value by 10;
            add 1 to exponent;
        end until;
```

图 7-17 浮点到 ASCII 码的转换算法

```
    else
        while value < 1 loop
            multiply value by 10;
            subtract 1 from exponent;
        end while;
    end if;
end if;

add 0.000005 to value;   { for rounding }
if value > 10
then
    divide value by 10;
    add 1 to exponent;
end if;
digit := int(value);        { truncate to integer }
convert digit to ASCII and store in destination string;
point at next destination byte;
store "." in destination string;
point at next destination byte;

for i := 1 to 5 loop
    value := 10 * (value - float(digit));
    digit := int(value);
    convert digit to ASCII and store in destination string;
    point at next destination byte;
end for;
store E in destination string;
point at next destination byte;
if exponent ≥ 0
then
    put + in destination string;
else
    put - in destination string;
    exponent := -exponent;
end if;
point at next destination byte;

convert exponent to two decimal digits;
convert two decimal digits of exponent to ASCII;
store characters of exponent in destination string;
```

图 7-17 （续）

小数点存储在目的字符串。然后由原数值减去之前所保留的整数部分，再用 10 乘以剩下的小数部分，截取新数的整数部分，这样重复操作，直到最后获得小数点后的五位整数。

在 ASCII 码字符串的小数部分产生之后，生成字母 E，指数用的加号或者减号，以及指数。指数最多包括两位数字——单精度的 IEEE 记数可以表示的最大数为 2^{128}，它小于 10^{39}。

图 7-18 是实现过程 ftoa 的设计代码，并有一个简短的测试驱动程序。过程需要传递两个参数，即要转换的浮点数和目的字符串的地址。

```
; floating point to ASCII code
; author:  R. Detmer
; revised:  8/2005
.386
.MODEL FLAT
ExitProcess PROTO NEAR32 stdcall, dwExitCode:DWORD
.STACK  4096              ; reserve 4096-byte stack
.DATA                     ; reserve storage for data
source      REAL4   145.8798
result      BYTE    12 DUP (?)

.CODE
_start:
            push  source        ; parameter 1, floating point value
            lea   eax, result   ; parameter 2, address of result
            push  eax
            call   ftoa         ; ftoa(source, result)
            INVOKE ExitProcess, 0
PUBLIC _start

; procedure ftoa(source, result)
; convert floating point number to ASCII string
; Parameters passed on the stack:
;   (1) 32-bit floating point value
;   (2) address of ASCII destination string
; ASCII string with format [blank/-]d.ddddd E[+/-]dd is generated.
; (The string is always 12 characters long )

C3 EQU 0100000000000000b
C2 EQU 0000010000000000b
C0 EQU 0000000100000000b

.DATA
value     REAL4  ?
ten       REAL4  10.0
one       REAL4  1.0
round     REAL4  0.000005
digit     DWORD  ?
exponent  DWORD  ?
controlWd WORD   ?
byteTen   BYTE   10

.CODE
ftoa      PROC NEAR32
          push ebp              ; establish stack frame
          mov  ebp, esp
          push eax              ; save registers
          push ebx
```

图 7-18 ftoa 过程和测试驱动程序

```
        push ecx
        push edi
        pushfd

        fstcw controlWd         ; get control word
        push controlWd          ; save control word
        or   controlWd, 0000110000000000b
        fldcw controlWd         ; set control to chop
        mov  edi, [ebp+8]       ; destination string address
        mov  eax, [ebp+12]      ; value to convert
        mov  exponent, 0        ; exponent := 0
        mov  value, eax         ; value to ST via memory
        fld  value
        ftst                    ; value >= 0?
        fstsw  ax               ; status word to AX
        and  ax, C0             ; check C0
        jnz  elseNeg            ; skip if set (value negative)
        mov  BYTE PTR [edi], ' '  ; blank for positive
        jmp  endifNeg
elseNeg: mov  BYTE PTR [edi], '-'  ; minus for negative
        fchs                    ; make number positive
endifNeg:
        inc  edi                ; point at next destination byte

        mov  exponent, 0        ; exponent := 0
        ftst                    ; value = 0?
        fstsw ax                ; status word to AX
        and  ax, C3             ; check C3
        jne  endifZero          ; skip if zero
        fcom ten                ; value > 10?
        fstsw ax                ; status word to AX
        and  ax, C3 or C2 or C0   ; check for all C3 = C2 = C0 = 0
        jnz  elseLess           ; skip if value not > 10
untilLess:
        fdiv ten                ; value := value/10
        inc  exponent           ; add 1 to exponent
        fcom ten                ; value < 10
        fstsw ax                ; status word to AX
        and  ax, C0             ; check C0
        jnz  untilLess          ; continue until value < 10
        jmp  endifBigger        ; exit if
elseLess:
whileLess:
        fcom one                ; value < 1
        fstsw ax                ; status word to AX
        and  ax, C0             ; check C0
        jz   endwhileLess       ; exit if not less
        fmul ten                ; value := 10*value
        dec  exponent           ; subtract 1 from exponent
        jmp  whileLess          ; continue while value < 1
endwhileLess:
endifBigger:
endifZero:
```

图 7-18 （续）

```
            fadd round              ; add rounding value
            fcom ten                ; value > 10?
            fstsw ax                ; status word to AX
            and  ax, C3 or C2 or C0  ; C3 = C2 = C0 = 0? (value > 10?)
            jnz  endifOver          ; skip if not
            fdiv ten                ; value := value/10
            inc  exponent           ; add 1 to exponent
endifOver:

; at this point 1.0 <= value < 10.0
            fist digit              ; store integer part
            mov  ebx, digit         ; copy integer to EBX
            or   ebx, 30h           ; convert digit to character
            mov  BYTE PTR [edi], bl  ; store character in destination
            inc  edi                ; point at next destination byte
            mov  BYTE PTR [edi], '.'     ; decimal point
            inc  edi                ; point at next destination byte

            mov  ecx, 5             ; count of remaining digits
forDigit:   fisub digit             ; subtract integer part
            fmul ten                ; multiply by 10
            fist digit              ; store integer part
            mov  ebx, digit         ; copy integer to BX
            or   ebx, 30h           ; convert digit to character
            mov  BYTE PTR [edi], bl  ; store character in destination
            inc  edi                ; point at next destination byte
            loop forDigit           ; repeat 5 times
            mov  BYTE PTR [edi], 'E'  ; exponent indicator
            inc  edi                ; point at next destination byte
            mov  eax, exponent      ; get exponent
            cmp  eax, 0             ; exponent >= 0 ?
            jnge NegExp
            mov  BYTE PTR [edi], '+'  ; non-negative exponent
            jmp  endifNegExp
NegExp:     mov  BYTE PTR [edi], '-'  ; negative exponent
            neg  ax                 ; change exponent to positive
endifNegExp:
            inc  edi                ; point at next destination byte

            div  byteTen            ; convert exponent to 2 digits
            or   eax, 3030h         ; convert both digits to ASCII
            mov  BYTE PTR [edi+1], ah  ; store characters in destination
            mov  BYTE PTR [edi], al

            pop  controlWd          ; restore control word
            fldcw controlWd
            popfd                   ; restore flags
            pop  edi                ; restore registers
            pop  ecx
            pop  ebx
            pop  eax
            pop  ebp
            ret  8                  ; return, removing parameters
ftoa        ENDP
            END
```

图 7-18 （续）

过程是通过名字引用控制位的指令开始的。C3、C2 和 C0 的控制位分别在第 14、10 和 8 位，并且都置“1”。对于 16 位的二进制数字，每个语句 EQU 为相应的符号名赋值。该过程也有自己的数据段，通过链接器，这个数据段与测试驱动程序所使用的数据段链接在一起。这也是一种在堆栈中存储局部变量的方法。

```
C3 EQU 0100000000000000b
C2 EQU 0000010000000000b
C0 EQU 0000000100000000b
```

在常规的过程开始语句后，FPU 控制字被复制到存储器中，并且让它进栈。这样，在过程结束后，它能被恢复，控制字的第 10、11 位用于控制四舍五入。接下来的两条指令将它们设为 11。这样，当一个浮点数存入整型存储器时，该数值的小数部分会舍去。

```
fstcw controlWd        ; get control word
push controlWd         ; save control word
or   controlWd, 0000110000000000b
fldcw controlWd        ; set control to chop
```

大部分过程的代码可直接实现设计，很容易理解；然而，对浮点型的比较运算需要作一些说明。第一段是：

```
ftst                   ; value >= 0?
fstsw  ax              ; status word to AX
and  ax, C0            ; check C0
jnz  elseNeg           ; skip if set (value negative)
```

ftst 指令比较 value 与 0.0 的值，并设置状态字中的标志位。为了测试这些位，状态字被复制给 AX。只有当 ST<0 时，C0 标志位置 1。除 C0 所对应的位外，and 指令屏蔽了其余所有的位。如果剩余位是非 0，表示 value 是负的，那么执行 jnz 指令。

判断是否“value>10”的程序和上述程序段类似，但更复杂，其代码如下：

```
fcom ten               ; value > 10?
fstsw ax               ; status word to AX
and  ax, C3 or C2 or C0    ; check for all C3 = C2 = C0 = 0
jnz  elseLess          ; skip if value not > 10
```

如果 ST> 操作数，那么 C3=C2=C0=0，三个控制位都为零。程序屏蔽 C3、C2 或 C0，可写为 0100010100000000。在汇编时（而不是执行时），or 运算将操作数组合在一起。

这里用了一种新方法将指数转换成两个 ASCII 码字符。执行下列指令时，在 AX 中的指数是非负的，并且小于 40。

```
div  byteTen           ; convert exponent to 2 digits
or   ax, 3030h         ; convert both digits to ASCII
mov  BYTE PTR [edi+1], ah   ; store characters in destination
mov  BYTE PTR [edi], al
```

将指数除以 10，商（十进制的高位）放入 AL，余数（低位）放入 AH。由 or 指令将商和余数同时转换成 ASCII 码，并保存在目的字符串中。

图 7-19 是程序退出之前的 Windbg 窗口。结果在存储器的地址 0x00404004 处显示。

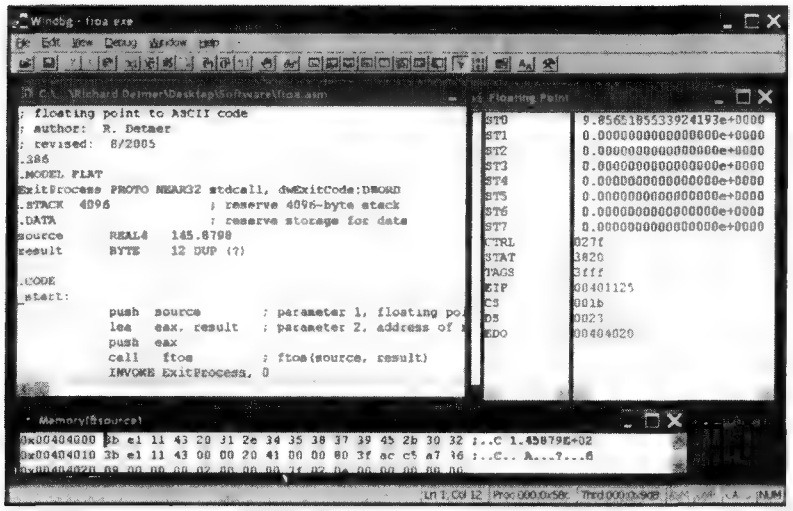


图 7-19 过程 ftoa 的执行

练习 7.3

1. 对于包含“1234”且没有其他字符的以“null”终止的字符串，过程 atof 将返回什么结果？换句话说，如果没有小数点，过程将如何处理？
2. 对于包含“12..34”且没有其他字符的以“null”终止的字符串，过程 atof 将返回什么结果？换句话说，如果有两个小数点，过程将如何处理？
3. 对于包含“--1234”且没有其他字符的以“null”终止的字符串，过程 atof 将返回什么结果？换句话说，如果是以两个负号开始，过程将如何处理？
4. 为什么过程 ftoa 能正确地产生 6 位有意义的十进制数和两位指数？

编程练习 7.3

1. 设计并实现一个 NEAR32 的过程 dtoa，它的功能与 ftoa 类似，只不过它是作用于双精度浮点值。目的字符串的格式是什么？（提示：见图 7-1）传递给堆栈的参数包含：（1）一个双精度浮点数的地址；（2）ASCII 码的目的字符串的地址。这些参数通过过程 dtoa 从堆栈中取出。过程将保存所有的寄存器，包括整数、标志位和浮点寄存器。

   为过程编写一个简单的测试驱动程序，并用 Windbg 查看结果。
2. 写一个 NEAR32 的过程 ftoa1，实现单精度浮点数和固定小数点格式的 ASCII 码字符串的转换。要特别说明的是，这个过程必须将下列 4 个参数压入堆栈：

   a. 一个单精度浮点值；

   b. 目的字符串的地址；

   c. 一个双字，表示生成的字符串的字符总个数 n；

   d. 一个双字，表示生成的小数点后的数字的个数 d。

   输出的字符串包含一个起始位的负号或空格、占用 $n-d-2$ 位（如果需要，起始位加入空格）的数值的整数部分、一个小数点，以及四舍五入后的数值存储了 d 位的小数部分。

为过程编写一个简单的测试驱动程序并用 Windbg 查看结果。

3. 下面的算法是计算一个实数 x 的近似立方根：

```
root := 1.0;
until (|root - oldRoot| < smallValue) loop
    oldRoot := root;
    root := (2.0 * root + x/(root * root)) / 3.0;
end until;
```

请编写一个 NEAR32 的过程 cuberoot 实现该设计，smallValue 的值是 0.001。假定有一个参数传递给堆栈，x 的值是一个单精度浮点数。过程将从堆栈中取出参数。在 ST 中返回结果。过程将保存所有寄存器，包含整数部分、标志位和浮点寄存器。

为过程编写一个简单的测试驱动程序，并用 Windbg 查看结果。

7.4 浮点数和嵌入式汇编

有些高级语言编译器能够翻译带嵌入式汇编代码（in-line assembly code）的程序，它允许大部分程序用高级语言编写，而小部分程序用汇编语言编写。这部分可能是优化的重点，或者可能实现一些高级语言不可能或很难实现的低级算法。

本节给出了一个程序例子，它使用微软 Visual Studio 2005 的 Visual C++ Win32 平台来编译。该例子执行与图 7-12 同样的计算，即求出两个浮点数平方和的平方根。但是，这个例子要求输入和输出数据，其中输入和输出语句用 C++ 语言编写。这里的汇编语言代码并不是作为一个过程使用。具体代码如图 7-20 所示。

```
// square root of sum of squares of two values
#include <iostream>
using namespace std;
int main()
{
  float value1;
  float value2;
  float sum;

  cout << "First value? ";
  cin >> value1;
  cout << "Second value? ";
  cin >> value2;

  __asm
  {
    fld    value1
    fld    st
    fmul
    fld    value2
    fld    st
    fmul
    fadd
    fsqrt
    fstp  sum
  }
  cout << "The sum is " << sum << endl;
  return 0;
}
```

图 7-20 嵌入的汇编语言代码

注意，对于这个编译程序，嵌入式汇编代码是以关键字 __asm 标志开始的（开头是两个下划线）。并且，汇编语言部分要用大括号括起来。注意，汇编语言语句可以引用 C++ 语句所声明的变量。最后，虽然这些汇编语言语句都是浮点型指令，但是任何语句都能用在嵌入式的汇编代码中，包含整数指令和带有标号的指令。

该程序的运行如图 7-21 所示。当在 Visual Studio 环境中运行该程序，将出现“Press any key to continue”的提示，按“Ctrl+F5”开始执行程序。

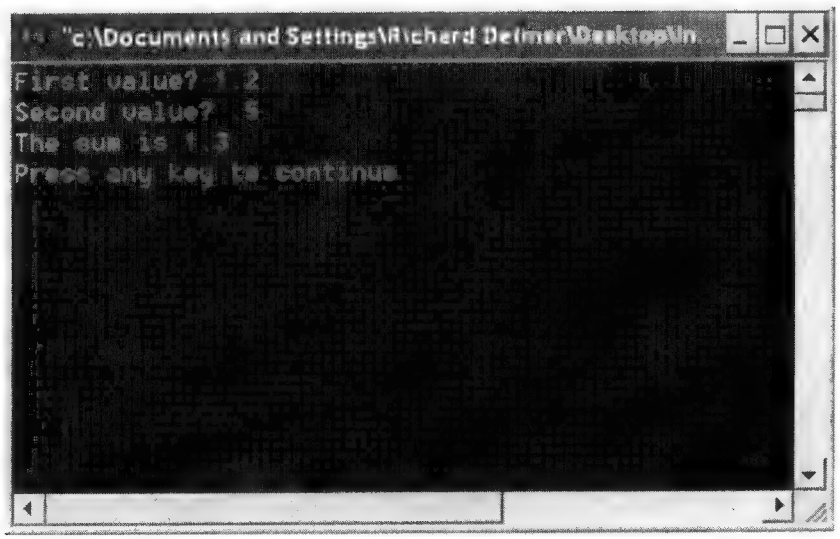


图 7-21 运行 C++/ 汇编语言的示例

编程练习 7.4

1. 设计一个完整的程序，它能提示用户输入一个十进制表示的圆的半径，并计算和显示圆的周长和面积。输入和输出用 C++ 语言编写，浮点数的计算用浮点数嵌入汇编语言的指令来编写。
2. 下面的算法是计算一个实数 x 的近似立方根：

```
root := 1.0;
until (|root - oldRoot| < smallValue) loop
    oldRoot := root;
    root := (2.0*root + x/(root*root)) / 3.0;
end until;
```

用 C++ 代码来声明变量，输入 x 的值并显示立方根 root。用嵌入的汇编语言代码计算立方根，smallValue 的值是 0.001。

7.5 本章小结

Intel 的 80x86 体系使用三种格式来表示浮点数：单精度（32 位）、双精度（64 位）和扩展实数（80 位）。每种格式包含一个符号位、指数部分和小数部分。

Intel 80x86 浮点运算器（FPU）有 8 个 80 位的数据寄存器，它们共同组成了一个堆栈。每个寄存器存储一个扩展实型浮点数。FPU 可以执行各种简单的载入、存储指令以及复杂的超越函数。比较指令可用来设置 FPU 状态寄存器中的位，这个状态字必须被复制到 AX 寄存器或者存储器，这样才能检查比较的结果。

浮点数和 ASCII 码之间的转换类似于前面讨论的 ASCII 码和整数的转换。最容易读的 ASCII 码格式是简单的十进制格式，而最简单的 ASCII 码格式用 E 记数法生成。

一些高级语言编译器可以翻译嵌入的汇编语言代码，其应用之一是和浮点型指令一起，用 C++ 这样的高级语言实现输入输出，而用汇编语言实现计算。而且，嵌入式汇编语言对其他关键或难以实现的应用也很有用。

附录 A　十六进制 /ASCII 码转换

00	NUL	(null)	20	space	40	@	60	`
01	SOH		21	!	41	A	61	a
02	STX		22	“	42	B	62	b
03	ETX		23	#	43	C	63	c
04	EOT		24	$	44	D	64	d
05	ENQ		25	%	45	E	65	e
06	ACK		26	&	46	F	66	f
07	BEL	(bell)	27	'	47	G	67	g
08	BS	(backspace)	28	(	48	H	68	h
09	HT	(tab)	29	)	49	I	69	i
0A	LF	(line feed)	2A	*	4A	J	6A	j
0B	VT		2B	+	4B	K	6B	k
0C	FF	(form feed)	2C	,	4C	L	6C	l
0D	CR	(return)	2D	-	4D	M	6D	m
0E	SO		2E	.	4E	N	6E	n
0F	SI		2F	/	4F	O	6F	o
10	DLE		30	0	50	P	70	p
11	DC1		31	1	51	Q	71	q
12	DC2		32	2	52	R	72	r
13	DC3		33	3	53	S	73	s
14	DC4		34	4	54	T	74	t
15	NAK		35	5	55	U	75	u
16	SYN		36	6	56	V	76	v
17	ETB		37	7	57	W	77	w
18	CAN		38	8	58	X	78	x
19	EM		39	9	59	Y	79	y
1A	SUB		3A	:	5A	Z	7A	z
1B	ESC	(“escape”)	3B	;	5B	[	7B	{
1C	FS		3C	<	5C	\	7C	|
1D	GS		3D	=	5D	]	7D	}
1E	RS		3E	>	5E	^	7E	~
1F	US		3F	?	5F	_	7F	DEL

附录 B　有用的 MS-DOS 命令

MS-DOS（和 Windows）使用类似 Unix 的分级文件结构。MS-DOS 的文件通过驱动器（C:、A:，等等）区分，驱动器后紧跟着路径以区分目录（文件夹），最后是文件名本身。例如，一个完整的文件名：A:\asm\project1\example.asm。符号“\”用于分隔路径中的组成部分与根目录的名字（顶级）。大多数 MS-DOS 系统设置显示当前的驱动器，以及作为提示符部分的路径（例如，C:\WINDOWS>）。

如果不具体指定路径中的驱动器或者目录，默认（default）指的是当前使用的驱动器或者目录。如要改变默认（当前）驱动器，只需敲一个新的驱动器字母和冒号就可以了。

如要改变默认（当前）目录，可用 CD 命令，符号“..”为到当前目录的父目录的快捷方式。例如，如果当前目录为 \Windows\Desktop，那么 CD.. 将改变当前目录到 \Windows。（注意：MS-DOS 不区分大小写，cd 同样可实现该功能。）

MD 命令创建一个新的目录。要在当前目录中创建一个新的目录，可用“MD *目录名*”来实现。

DIR 命令显示当前文件夹中的文件目录。另外，DIR 命令也可给出所想要的目录的路径。如，DIR C:\projects。可用 * 作为一个通配符。例如，DIR s*.* 查找所有名字以字母 s 开头的文件。

COPY 命令将文件从一个目录复制到另外一个目录。格式为“COPY *源目的*”。如果不具体指定一个目的文件的名字，那么，目的文件名将采用源文件名。也可用 COPY 命令在同一个目录中创建一个文件的副本，但要用不同的文件名。COPY 命令允许使用通配符 * 来复制一组文件。

EDIT 命令用于创建或者修改一个文本(text)文件。EDIT 文件名调用一个简单的文本编辑器，如果该文件名的文件存在，则打开该文件。如果该文件名的文件不存在，则创建该文件。EDIT 自身带有帮助系统可提供很多信息，这些信息要比所需求的多得多。

REN 命令用来给文件重命名。格式为“REN *旧文件名 新文件名*”。

通过敲入命令“/?”，可获得大多数命令的更多信息。

注意：如果在 MS-DOS 下正在做某件事情，这并不意味着不能使用其他的 Windows 工具。可用“我的电脑”或者“Explorer”来创建目录、复制文件、重命名文件等等。也可用记事本来编辑文件，但是要注意的是，记事本通常对每一个文件加上扩展名 .TXT。通常，要避免使用字处理程序来编辑像汇编语言源代码文件那样的文本文件，如果确实要用，最好用简单的文本格式保存文件。

附录 C　MASM 6.11 保留字

AAA
AAD
AAM
AAS
ABS
ADC
ADD
AH
AL
ALIGN
.ALPHA
AND
ARPL
ASSUME
AT
AX
BH
BL
BOUND
BP
.BREAK
BSF
BSR
BSWAP
BT
BTC
BTR
BTS
BX
BYTE
CALL
CARRY?
CASEMAP
CATSTR
@CatStr
CBW
CDQ
CH
CL
CLC
CLD
CLI
CLTS
CMC
CMP
CMPS
CMPSB
CMPSD
CMPSW
CMPXCHG
.CODE
@code
@CodeSize
COMM
COMMENT
COMMON
CONST
.CONTINUE
@Cpu
@CREF
CS
@CurSeg
CWD
CWDE
CX
DAA
DAS
.DATA
@data
.DATA?
@DataSize
@Date
DEC
DH
DI
DIV
DL
.DOSSEG
DOTNAME
DS
DUP
DWORD
DX
EAX
EBP
EBX
ECHO
ECX
EDI
EDX
ELSE
ELSEIF
ELSEIFDIF
ELSEIFIDN
EMULATOR
END
ENDIF
.ENDIF
ENDM
ENDP
ENDS
ENDW
ENTER
@Environment
EPILOGUE
EQ
EQU
ERR
.ERRB
ERRDEF
.ERRDIF
.ERRE
.ERRIDN
.ERRNB
FCOMP
FCOMPP
FCOS
FDECSTP
FDISI
FDIV
FDIVP
FDIVR
FDIVRP
FENI
FLDPI
FLDZ
FMUL
FMULP
FNCLEX
FNDISI
FNENI
FNINIT
FNOP
FNSAVE
FST
FSTCW
FSTENV
FSTENVD
FSTENVW
FSTP
FSTSW
FSUB
FSUBP
FSUBR

ERRNDEF
.ERRNZ
ESI
ES
ESP
EVEN
.EXIT
EXITM
EXPORT
EXPR16
EXPR32
EXTERN
EXTERNDEF
@F
F2XM1
FABS
FADD
FADDP
FARDATA
@fardata
FARDATA?
@fardata?
FBLD
FBSTP
FCHS
FCLEX
FCOM
FFREE
FIADD
FICOM
FICOMP
FIDIV
FIDIVR
FILD
@FileCur
@FileName
FIMUL
FINCSTP
FINIT
FIST
FISTP
FISUB
FISUBR
FLAT
FLD
FLD1
FLDCW
FLDENV
FLDENVD
FLDENVW
FLDL2E
FLDL2T
FLDLG2
FLDLN2
FNSAVED
FNSAVEW
FNSTCW
FNSTENV
FNSTENVD
FNSTENVW
FNSTSW
FOR
FORC
FORCEFRAME
FPATAN
FPREM
FPREM1
FPTAN
FRNDINT
FRSTOR
FRSTORD
FRSTORW
FS
FSAVE
FSAVED
FSAVEW
FSCALE
FSETPM
FSIN
FSINCOS
FSQRT
FSUBRP
FTST
FUCOM
FUCOMP
FUCOMPP
FWAIT
FWORD
FXAM
FXCH
FXTRACT
FYL2X
FYL2XP1
GE
GOTO
GROUP
GS
GT
HIGH
HIGHWORD
HLT
IDIV
IF
.IF
IFB
IFDEF
IFDIF
IFDIFI
IFE
IFIDN
IFIDNI
IFNB
IFNDEF
IMUL
IN
INC
INCLUDE
INCLUDELIB
INS
INSB
INSD
INSTR
@InStr
INSW
INT
INTO
JNAE
JNB
JNBE
JNC
JNE
JNG
JNGE
JNL
JNLE
JNO
JNP
JNS
JNZ
JO
JP
JPE
JPO
JS
.LISTIF
.LISTMACRO
.LISTMACROALL
LJMP
LLDT
LMSW
LOADDS
LOCAL
LOCK
LODS
LODSB
LODSD
LODSW
LOOP
LOOPD
LOOPW
LOW
LOWWORD
MUL
NE
NEG
.NO87
.NOCREF
NODOTNAME
NOKEYWORD
.NOLIST
.NOLISTIF
.NOLISTMACRO
NOLJMP
NOM510
NOP
NOREADONLY
NOSCOPED
NOSIGNEXTEND
NOT
OFFSET

INVD
INVLPG
INVOKE
IRET
IRETD
JA
JAE
JB
JBE
JC
JCXZ
JE
JECXZ
JG
JGE
JL
JLE
JMP
JNA
PROC
PROLOGUE
PROTO
PTR
PUBLIC
PURGE
PUSH
PUSHA
PUSHAD
PUSHCONTEXT
PUSHD
PUSHF
PUSHFD
PUSHW
QWORD
.RADIX
RCL
RCR
READONLY
REAL10
REAL4
REAL8
RECORD
REP
REPE
JZ
LABEL
LAHF
LANGUAGE
LAR
LDS
LE
LEA
LEAVE
LENGTH
LENGTHOF
LES
LFS
LGDT
LGS
LIDT
@Line
.LIST
.LISTALL
RET
RETF
RETN
ROL
ROR
SAHF
SAL
SAR
SBB
SBYTE
SCAS
SCASB
SCASD
SCASW
SCOPED
SDWORD
SEG
SEGMENT
.SEQ
SET
.SETIF2
SGDT
SHL
SHLD
SHORT
LROFFSET
LSL
LSS
LT
LTR
M510
MACRO
MASK
MEMORY
MOD
.MODEL
@Model
MOV
MOVS
MOVSB
MOVSD
MOVSW
MOVSX
MOVZX
SIGN?
SIZEOF
SIZESTR
@SizeStr
SLDT
SMSW
SP
SS
.STACK
@stack
.STARTUP
STC
STD
STDCALL
STI
STOS
STOSB
STOSD
STOSW
STR
STRUCT
SUB
SUBSTR
@SubStr
SUBTITLE
OPTION
OR
ORG
OUT
OUTS
OUTSB
OUTSD
OUTSW
OVERFLOW?
PAGE
PARA
PARITY?
POP
POPA
POPAD
POPCONTEXT
POPF
POPFD
PRIVATE
TEXTEQU
.TFCOND
THIS
@Time
TITLE
TYPE
TYPEDEF
UNION
.UNTIL
USE16
USE32
USES
VERR
@Version
VERW
WAIT
WBINVD
WHILE
.WHILE
WIDTH
WORD
@WordSize
XADD
XCHG
XLAT

REPEAT	SHR	SWORD	XOR
REPNE	SHRD	SYSCALL	ZERO?
REPNZ	SI	TBYTE	
REPZ	SIDT	TEST	

附录 D　80x86 指令（按助记符排列）

助记符	操作数	受影响的标志位[⊖]	操作码	字节数
aaa	无	AF, CF SF, ZF, OF, PF ?	37	1
aad	无	SF, ZF, PF OF, AF, CF ?	D5 0A	2
aam	无	SF, ZF, PF OF, AF, CF ?	D4 0A	2
aas	无	AF, CF SF, ZF, OF, PF ?	3F	1
adc	AL, imm8	SF, ZF, OF, CF, PF, AF	14	2
adc	AX, imm16 EAX, imm32	SF, ZF, OF, CF, PF, AF	15	3 5
adc	reg8, imm8	SF, ZF, OF, CF, PF, AF	80	3
adc	reg16, imm16 reg32, imm32	SF, ZF, OF, CF, PF, AF	81	4 6
adc	reg16, imm8 reg32, imm8	SF, ZF, OF, CF, PF, AF	83	3
adc	mem8, imm8	SF, ZF, OF, CF, PF, AF	80	3+
adc	mem16, imm16 mem32, imm32	SF, ZF, OF, CF, PF, AF	81	4+ 6+
adc	mem16, imm8 mem32, imm8	SF, ZF, OF, CF, PF, AF	83	3+
adc	reg8, reg8	SF, ZF, OF, CF, PF, AF	12	2
adc	reg16, reg16 reg32, reg32	SF, ZF, OF, CF, PF, AF	13	2
adc	reg8, mem8	SF, ZF, OF, CF, PF, AF	12	2+
adc	reg16, mem16 reg32, mem32	SF, ZF, OF, CF, PF, AF	13	2+
adc	mem8, reg8	SF, ZF, OF, CF, PF, AF	10	2+
adc	mem16, reg16 mem32, reg32	SF, ZF, OF, CF, PF, AF	11	2+

⊖ 问号？表示标志位值可能改变，不过这些值没有意义。

助记符	操作数	受影响的标志位	操作码	字节数
add	AL, imm8	SF, ZF, OF, CF, PF, AF	04	2
add	AX, imm16 EAX, imm32	SF, ZF, OF, CF, PF, AF	05	3 5
add	reg8, imm8	SF, ZF, OF, CF, PF, AF	80	3
add	reg16, imm16 reg32, imm32	SF, ZF, OF, CF, PF, AF	81	4 6
add	reg16, imm8 reg32, imm8	SF, ZF, OF, CF, PF, AF	83	3
add	mem8, imm8	SF, ZF, OF, CF, PF, AF	80	3+
add	mem16, imm16 mem32, imm32	SF, ZF, OF, CF, PF, AF	81	4+ 6+
add	mem16, imm8 mem32, imm8	SF, ZF, OF, CF, PF, AF	83	3+
add	reg8, reg8	SF, ZF, OF, CF, PF, AF	02	2
add	reg16, reg16 reg32, reg32	SF, ZF, OF, CF, PF, AF	03	2
add	reg8, mem8	SF, ZF, OF, CF, PF, AF	02	2+
add	reg16, mem16 reg32, mem32	SF, ZF, OF, CF, PF, AF	03	2+
add	mem8, reg8	SF, ZF, OF, CF, PF, AF	00	2+
add	mem16, reg16 mem32, reg32	SF, ZF, OF, CF, PF, AF	01	2+
and	AL, imm8	SF, ZF, OF, CF, PF, AF	24	2
and	AX, imm16 EAX, imm32	SF, ZF, OF, CF, PF, AF	25	3 5
and	reg8, imm8	SF, ZF, OF, CF, PF, AF	80	3
and	reg16, imm16 reg32, imm32	SF, ZF, OF, CF, PF, AF	81	4 6
and	reg16, imm8 reg32, imm8	SF, ZF, OF, CF, PF, AF	83	3
and	mem8, imm8	SF, ZF, OF, CF, PF, AF	80	3+
and	mem16, imm16 mem32, imm32	SF, ZF, OF, CF, PF, AF	81	4+ 6+
and	mem16, imm8 mem32, imm8	SF, ZF, OF, CF, PF, AF	83	3+
and	reg8, reg8	SF, ZF, OF, CF, PF, AF	22	2
and	reg16, reg16 reg32, reg32	SF, ZF, OF, CF, PF, AF	23	2
and	reg8, mem8	SF, ZF, OF, CF, PF, AF	22	2+
and	reg16, mem16 reg32, mem32	SF, ZF, OF, CF, PF, AF	23	2+

助记符	操作数	受影响的标志位	操作码	字节数
and	mem8, reg8	SF, ZF, OF, CF, PF, AF	20	2+
and	mem16, reg16 mem32, reg32	SF, ZF, OF, CF, PF, AF	21	2+
call	rel32	无	E8	5
call	reg32 (near indirect)	无	FF	2
call	mem32 (near indirect)	无	FF	2+
call	far direct	无	9A	7
call	far indirect	无	FF	6
cbw	无	无	98	1
cdq	无	无	99	1
clc	无	CF	F8	1
cld	无	DF	FC	1
cmc	无	CF	F5	1
cmp	AL, imm8	SF, ZF, OF, CF, PF, AF	3C	2
cmp	AX, imm16 EAX, imm32	SF, ZF, OF, CF, PF, AF	3D	3 5
cmp	reg8, imm8	SF, ZF, OF, CF, PF, AF	80	3
cmp	reg16, imm16 reg32, imm32	SF, ZF, OF, CF, PF, AF	81	4 6
cmp	reg16, imm8 reg32, imm8	SF, ZF, OF, CF, PF, AF	83	3
cmp	mem8, imm8	SF, ZF, OF, CF, PF, AF	80	3+
cmp	mem16, imm16 mem32, imm32	SF, ZF, OF, CF, PF, AF	81	4+ 6+
cmp	mem16, imm8 mem32, imm8	SF, ZF, OF, CF, PF, AF	83	3+
cmp	reg8, reg8	SF, ZF, OF, CF, PF, AF	38	2
cmp	reg16, reg16 reg32, reg32	SF, ZF, OF, CF, PF, AF	3B	2
cmp	reg8, mem8	SF, ZF, OF, CF, PF, AF	3A	2+
cmp	reg16, mem16 reg32, mem32	SF, ZF, OF, CF, PF, AF	3B	2+
cmp	mem8, reg8	SF, ZF, OF, CF, PF, AF	38	2+
cmp	mem16, reg16 mem32, reg32	SF, ZF, OF, CF, PF, AF	39	2+
cmpsb	无	无	A6	1
cmpsw cmpsd	无	无	A7	1
cwd	无	无	99	1

助记符	操作数	受影响的标志位	操作码	字节数
cwde	无	无	98	1
daa	无	SF, ZF, PF, AF OF ?	27	1
das	无	SF, ZF, PF, AF OF ?	2F	1
dec	reg8		FE	2
dec	AX EAX	SF, ZF, OF, PF, AF	48	1
dec	CX ECX	SF, ZF, OF, PF, AF	49	1
dec	DX EDX	SF, ZF, OF, PF, AF	4A	1
dec	BX EBX	SF, ZF, OF, PF, AF	4B	1
dec	SP ESP	SF, ZF, OF, PF, AF	4C	1
dec	BP EBP	SF, ZF, OF, PF, AF	4D	1
dec	SI ESI	SF, ZF, OF, PF, AF	4E	1
dec	DI EDI	SF, ZF, OF, PF, AF	4F	1
dec	mem8	SF, ZF, OF, PF, AF	FE	2+
dec	mem16 mem32	SF, ZF, OF, PF, AF	FF	2+
div	reg8	SF, ZF, OF, PF, AF ?	F6	2
div	reg16 reg32	SF, ZF, OF, PF, AF ?	F7	2
div	mem8	SF, ZF, OF, PF, AF ?	F6	2+
div	mem16 mem32	SF, ZF, OF, PF, AF ?	F7	2+
idiv	reg8	SF, ZF, OF, PF, AF ?	F6	2
idiv	reg16 reg32	SF, ZF, OF, PF, AF ?	F7	2
idiv	mem8	SF, ZF, OF, PF, AF ?	F6	2+
idiv	mem16 mem32	SF, ZF, OF, PF, AF ?	F7	2+
imul	reg8	OF, CF SF, ZF, PF, AF ?	F6	2
imul	reg16 reg32	OF, CF SF, ZF, PF, AF ?	F7	2

助记符	操作数	受影响的标志位	操作码	字节数
imul	mem8	OF, CF SF, ZF, PF, AF ?	F6	2+
imul	mem16 mem32	OF, CF SF, ZF, PF, AF ?	F7	2+
imul	reg16, reg16 reg32, reg32	OF, CF SF, ZF, PF, AF ?	0F AF	3
imul	reg16, mem16 reg32, mem32	OF, CF SF, ZF, PF, AF ?	0F AF	3+
imul	reg16, imm8 reg32, imm8	OF, CF SF, ZF, PF, AF ?	6B	3
imul	mem16 mem32	OF, CF SF, ZF, PF, AF ?	F7	4 6
imul	reg16, reg16, imm8 reg32, reg32, imm8	OF, CF SF, ZF, PF, AF ?	6B	3
imul	reg16, reg16, imm16 reg32, reg32, imm32	OF, CF SF, ZF, PF, AF ?	69	4 6
imul	reg16, mem16, imm8 reg32, mem32, imm8	OF, CF SF, ZF, PF, AF ?	6B	3+
imul	reg16, mem16, imm16 reg32, mem32, imm32	OF, CF SF, ZF, PF, AF ?	69	4+ 6+
inc	reg8	SF, ZF, OF, PF, AF	FE	2
inc	AX EAX	SF, ZF, OF, PF, AF	40	1
inc	CX ECX	SF, ZF, OF, PF, AF	41	1
inc	DX EDX	SF, ZF, OF, PF, AF	42	1
inc	BX EBX	SF, ZF, OF, PF, AF	43	1
inc	SP ESP	SF, ZF, OF, PF, AF	44	1
inc	BP EBP	SF, ZF, OF, PF, AF	45	1
inc	SI ESI	SF, ZF, OF, PF, AF	47	1
inc	DI EDI	SF, ZF, OF, PF, AF	48	1
inc	mem8	SF, ZF, OF, PF, AF	FE	2+
inc	mem16 mem32	SF, ZF, OF, PF, AF	FF	2+
ja jnbe	rel8	无	77	7+, 3

助记符	操作数	受影响的标志位	操作码	字节数
ja jnbe	rel32	无	0F 87	7+, 3
jae jnb	rel8	无	73	7+, 3
jae jnb	rel32	无	0F 83	7+, 3
jb jnae	rel8	无	72	7+, 3
jb jnae	rel32	无	0F 82	7+, 3
jbe jna	rel8	无	76	7+, 3
jbe jna	rel32	无	0F 86	7+, 3
jc	rel8	无	72	7+, 3
jc	rel32	无	0F 82	7+, 3
je jz	rel8	无	74	7+, 3
je jz	rel32	无	0F 84	7+, 3
jecxz	rel8	无	E3	2
jg jnle	rel8	无	7F	7+, 3
jg jnle	rel32	无	0F 8F	7+, 3
jge jnl	rel8	无	7D	7+, 3
jge jnl	rel32	无	0F 8D	7+, 3
jl jnge	rel8	无	7C	7+, 3
jl jnge	rel32	无	0F 8C	7+, 3
jle jng	rel8	无	7E	7+, 3
jle jng	rel32	无	0F 8E	7+, 3
jmp	rel8	无	EB	2
jmp	rel32	无	E9	5
jmp	reg32	无	FF	2
jmp	mem32	无	FF	2+

助记符	操作数	受影响的标志位	操作码	字节数
jnc	rel8	无	73	7+, 3
jnc	rel32	无	0F 83	7+, 3
jne jnz	rel8	无	75	7+, 3
jne jnz	rel32	无	0F 85	7+, 3
jno	rel8	无	71	7+, 3
jno	rel32	无	0F 81	7+, 3
jnp jpo	rel8	无	7B	7+, 3
jnp jpo	rel32	无	0F 8B	7+, 3
jns	rel8	无	79	7+, 3
jns	rel32	无	0F 89	7+, 3
jo	rel8	无	70	7+, 3
jo	rel32	无	0F 80	7+, 3
jp jpe	rel8	无	7A	7+, 3
jp jpe	rel32	无	0F 8A	7+, 3
js	rel8	无	78	7+, 3
js	rel32	无	0F 88	7+, 3
lea	reg32, mem32	无	8D	2+
lodsb	无	无	AC	1
lodsw lodsd	无	无	AD	1
loop	无	无	E2	11+
loope loopz	无	无	E1	11+
loopne loopnz	无	无	E0	11+
mov	AL, imm8	无	B0	2
mov	CL, imm8	无	B1	2
mov	DL, imm8	无	B2	2
mov	BL, imm8	无	B3	2
mov	AH, imm8	无	B4	2
mov	CH, imm8	无	B5	2
mov	DH, imm8	无	B6	2
mov	BH, imm8	无	B7	2

助记符	操作数	受影响的标志位	操作码	字节数
mov	AX, imm16 EAX, imm32	无	B8	3 5
mov	CX, imm16 ECX, imm32	无	B9	3 5
mov	DX, imm16 EDX, imm32	无	BA	3 5
mov	BX, imm16 EBX, imm32	无	BB	3 5
mov	SP, imm16 ESP, imm32	无	BC	3 5
mov	BP, imm16 EPB, imm32	无	BD	3 5
mov	SI, imm16 ESI, imm32	无	BE	3 5
mov	DI, imm16 EDI, imm32	无	BF	3 5
mov	mem8, imm8	无	C6	3+
mov	mem16, imm16 mem32, imm32	无	C7	4+ 6+
mov	reg8, reg8	无	8A	2
mov	reg16, reg16 reg32, reg32	无	8B	2
mov	AL, direct	无	A0	5
mov	AX, direct EAX, direct	无	A1	5
mov	reg8, mem8	无	8A	2+
mov	reg16, mem16 reg32, mem32	无	8B	2+
mov	mem8, reg8	无	88	2+
mov	mem16, reg16 mem32, reg32	无	89	2+
mov	direct , AL	无	A2	5
mov	direct, AX direct, EAX	无	A3	5
mov	sreg, reg16	无	8E	2
mov	reg16, sreg	无	8C	2
mov	sreg, mem16	无	8E	2+
mov	mem16, sreg	无	8C	2+
movsb	无	无	A4	1
movsw	无	无	A5	1

助记符	操作数	受影响的标志位	操作码	字节数
movsd				
movsx	reg16, reg8 reg32, reg8	无	0F BE	3
movsx	reg16, mem8 reg32, mem8	无	0F BE	3+
movsx	reg32, reg16	无	0F BF	3
movsx	reg32, mem16	无	0F BF	3+
movzx	reg16, reg8 reg32, reg8	无	0F B6	3
movzx	reg16, mem8 reg32, mem8	无	0F B6	3+
movzx	reg32, reg16	无	0F B7	3
movzx	reg32, mem16	无	0F B7	3+
mul	reg8	OF, CF SF, ZF, PF, AF ?	F6	2
mul	reg16 reg32	OF, CF SF, ZF, PF, AF ?	F7	2
mul	mem8	OF, CF SF, ZF, PF, AF ?	F6	2+
mul	mem16 mem32	OF, CF SF, ZF, PF, AF ?	F7	2+
neg	reg8	SF, ZF, OF, CF, PF, AF	F6	2
neg	reg16 reg32	SF, ZF, OF, CF, PF, AF	F7	2
neg	mem8	SF, ZF, OF, CF, PF, AF	F6	2+
neg	mem16 mem32	SF, ZF, OF, CF, PF, AF	F7	2+
not	reg8	无	F6	2
not	reg16 reg32	无	F7	2
not	mem8	无	F6	2+
not	mem16 mem32	无	F7	2+
or	AL, imm8	SF, ZF, OF, CF, PF, AF	0C	2
or	AX, imm16 EAX, imm32	SF, ZF, OF, CF, PF, AF	0D	3 5
or	reg8, imm8	SF, ZF, OF, CF, PF, AF	80	3
or	reg16, imm16 reg32, imm32	SF, ZF, OF, CF, PF, AF	81	4 6

助记符	操作数	受影响的标志位	操作码	字节数
or	reg16, imm8 reg32, imm8	SF, ZF, OF, CF, PF, AF	83	3
or	mem8, imm8	SF, ZF, OF, CF, PF, AF	80	3+
or	mem16, imm16 mem32, imm32	SF, ZF, OF, CF, PF, AF	81	4+ 6+
or	mem16, imm8 mem32, imm8	SF, ZF, OF, CF, PF, AF	83	3+
or	reg8, reg8	SF, ZF, OF, CF, PF, AF	0A	2
or	reg16, reg16 reg32, reg32	SF, ZF, OF, CF, PF, AF	0B	2
or	reg8, mem8	SF, ZF, OF, CF, PF, AF	0A	2+
or	reg16, mem16 reg32, mem32	SF, ZF, OF, CF, PF, AF	0B	2+
or	mem8, reg8	SF, ZF, OF, CF, PF, AF	08	2+
or	mem16, reg16 mem32, reg32	SF, ZF, OF, CF, PF, AF	09	2+
pop	AX EAX	无	58	1
pop	CX ECX	无	59	1
pop	DX EDX	无	5A	1
pop	BX EBX	无	5B	1
pop	SP ESP	无	5C	1
pop	BP EBP	无	5D	1
pop	SI ESI	无	5E	1
pop	DI EDI	无	5F	1
pop	DS	无	1F	1
pop	ES	无	07	1
pop	SS	无	17	1
pop	FS	无	0F A1	2
pop	GS	无	0F A9	2
pop	mem16 mem32	无	8F	2+
popa	无	无	61	1

助记符	操作数	受影响的标志位	操作码	字节数
popad				
popf popfd	无	无	9D	1
push	AX EAX	无	50	1
push	CX ECX	无	51	1
push	DX EDX	无	52	1
push	BX EBX	无	53	1
push	SP ESP	无	54	1
push	BP EBP	无	55	1
push	SI ESI	无	56	1
push	DI EDI	无	57	1
push	CS	无	0E	1
push	DS	无	1E	1
push	ES	无	06	1
push	SS	无	16	1
push	FS	无	0F A0	2
push	GS	无	0F A8	2
push	mem16 mem32	无	FF	2+
push	imm8	无	6A	2
push	imm16 imm32	无	68	3 5
pusha pushad	无	无	60	1
pushf pushfd	无	无	9C	1
rep repz repe	无 (string instruction prefix)	无	F3	1
rep movsb	无	无	F3 A4	2
rep movsw	无	无	F3 A5	2

助记符	操作数	受影响的标志位	操作码	字节数
rep movsd				
rep stosb	无	无	F3 A6	2
rep stosw rep stosd	无	无	F3 A7	2
repe cmpsb	无	无	F3 A6	2
repe cmpsw repe cmpsd	无	无	F3 A7	2
repe scasb	无	无	F3 AE	2
repe scasw repe scasd	无	无	F3 AF	2
repne cmpsb	无	无	F2 A6	2
repne cmpsw repne cmpsd	无	无	F2 A7	2
repne scasb	无	无	F2 AE	2
repne scasw repne scasd	无	无	F2 AF	2
repnz repne	无 (string instruction prefix)	无	F2	1
ret (far)	无	无	CB	1
ret (far)	imm16	无	CA	3
ret (near)	无	无	C3	1
ret (near)	imm16	无	C2	3
rol ror	reg8	SF, ZF, OF, CF, PF AF ?	D0	2
rol ror	reg16 reg32	SF, ZF, OF, CF, PF AF ?	D1	2
rol ror	mem8	SF, ZF, OF, CF, PF AF ?	D0	2+
rol ror	reg16 reg32	SF, ZF, OF, CF, PF AF ?	D1	2+

助记符	操作数	受影响的标志位	操作码	字节数
rol ror	reg8, imm8	SF, ZF, OF, CF, PF AF ?	C0	3
rol ror	reg16, imm8 reg32, imm8	SF, ZF, OF, CF, PF AF ?	C1	3
rol ror	mem8, imm8	SF, ZF, OF, CF, PF AF ?	C0	3+
rol ror	mem16, imm8 mem32,imm8	SF, ZF, OF, CF, PF AF ?	C1	3+
rol ror	reg8, CL	SF, ZF, OF, CF, PF AF?	D2	2
rol ror	reg16, CL reg32, CL	SF, ZF, OF, CF, PF AF ?	D3	2
rol ror	mem8, CL	SF, ZF, OF, CF, PF AF ?	D2	2+
rol ror	mem16, CL mem32, CL	SF, ZF, OF, CF, PF AF ?	D3	2+
sbb	AL, imm8	SF, ZF, OF, CF, PF, AF	1C	2
sbb	AX, imm16 EAX, imm32	SF, ZF, OF, CF, PF, AF	1D	3 5
sbb	reg8, imm8	SF, ZF, OF, CF, PF, AF	80	3
sbb	reg16, imm16 reg32, imm32	SF, ZF, OF, CF, PF, AF	81	4 6
sbb	reg16, imm8 reg32, imm8	SF, ZF, OF, CF, PF, AF	83	3
sbb	mem8, imm8	SF, ZF, OF, CF, PF, AF	80	3+
sbb	mem16, imm16 mem32, imm32	SF, ZF, OF, CF, PF, AF	81	4+ 6+
sbb	mem16, imm8 mem32, imm8	SF, ZF, OF, CF, PF, AF	83	3+
sbb	reg8, reg8	SF, ZF, OF, CF, PF, AF	1A	2
sbb	reg16, reg16 reg32, reg32	SF, ZF, OF, CF, PF, AF	1B	2
sbb	reg8, mem8	SF, ZF, OF, CF, PF, AF	1A	2+
sbb	reg16, mem16 reg32, mem32	SF, ZF, OF, CF, PF, AF	1B	2+
sbb	mem8, reg8	SF, ZF, OF, CF, PF, AF	18	2+
sbb	mem16, reg16 mem32, reg32	SF, ZF, OF, CF, PF, AF	19	2+
scasb	无	无	AE	1
scasw scasd	无	无	AE	1

助记符	操作数	受影响的标志位	操作码	字节数
shl/sal shr sar	reg8	SF, ZF, OF, CF, PF AF ?	D0	1
shl/sal shr sar	reg16 reg32	SF, ZF, OF, CF, PF AF ?	D1	2
shl/sal shr sar	mem8	SF, ZF, OF, CF, PF AF ?	D0	2+
shl/sal shr	reg16 reg32	SF, ZF, OF, CF, PF AF ?	D1	2+
shl/sal shr sar	reg8, imm8	SF, ZP, OF, CF, PF AF?	C0	3
shl/sal shr sar	reg16, imm8 reg32, imm8	SF, ZF, OF, CF, PF AF ?	C1	3
shl/sal shr sar	mem8, imm8	SF, ZF, OF, CF, PF AF ?	C0	3+
shl/sal shr sar	mem16, imm8 mem32, imm8	SF, ZF, OF, CF, PF AF ?	C1	3+
shl/sal shr sar	reg8, CL	SF, ZF, OF, CF, PF AF ?	D2	2
shl/sal shr sar	reg16, CL reg32, CL	SF, ZF, OF, CF, PF AF ?	D3	2
shl/sal shr sar	mem8, CL	SF, ZF, OF, CF, PF AF ?	D2	2+
shl/sal shr sar	mem16, CL mem32, CL	SF, ZF, OF, CF, PF AF ?	D3	2+
shld	reg16, reg16, imm8 reg32, reg32, imm8	SF, ZF, CF, PF OF, AF ?	0F 04	4
shld	mem16, reg16, imm8 mem32, reg32, imm8	SF, ZF, CF, PF OF, AF ?	0F 04	4+
shld	reg16, reg16, CL reg32, reg32, CL	SF, ZF, CF, PF OF, AF ?	0F 05	3
shld	mem16, reg16, CL mem32, reg32, CL	SF, ZF, CF, PF OF, AF ?	0F 05	3+
shrd	reg16, reg16, imm8	SF, ZF, CF, PF	0F AC	4

助记符	操作数	受影响的标志位	操作码	字节数
	reg32, reg32, imm8	OF, AF ?		
shrd	mem16, reg16, imm8 mem32, reg32, imm8	SF, ZF, CF, PF OF, AF ?	0F AC	4+
shrd	reg16, reg16, CL reg32, reg32, CL	SF, ZF, CF, PF OF, AF ?	0F AD	3
shrd	mem16, reg16, CL mem32, reg32, CL	SF, ZF, CF, PF OF, AF ?	0F AD	3+
stc	无	CF	F9	1
std	无	DF	FD	1
stosb	无	无	AA	1
stosw stosd	无	无	AB	1
sub	AL, imm8	SF, ZF, OF, CF, PF, AF	2C	2
sub	AX, imm16 EAX, imm32	SF, ZF, OF, CF, PF, AF	2D	3 5
sub	reg8, imm8	SF, ZF, OF, CF, PF, AF	80	3
sub	reg16, imm16 reg32, imm32	SF, ZF, OF, CF, PF, AF	81	4 6
sub	reg16, imm8 reg32, imm8	SF, ZF, OF, CF, PF, AF	83	3
sub	mem8, imm8	SF, ZF, OF, CF, PF, AF	80	3+
sub	mem16, imm16 mem32, imm32	SF, ZF, OF, CF, PF, AF	81	4+ 6+
sub	mem16, imm8 mem32, imm8	SF, ZF, OF, CF, PF, AF	83	3+
sub	reg8, reg8	SF, ZF, OF, CF, PF, AF	2A	2
sub	reg16, reg16 reg32, reg32	SF, ZF, OF, CF, PF, AF	2B	2
sub	reg8, mem8	SF, ZF, OF, CF, PF, AF	2A	2+
sub	reg16, mem16 reg32, mem32	SF, ZF, OF, CF, PF, AF	2B	2+
sub	mem8, reg8	SF, ZF, OF, CF, PF, AF	28	2+
sub	mem16, reg16 mem32, reg32	SF, ZF, OF, CF, PF, AF	29	2+
test	AL, imm8	SF, ZF, OF, CF, PF, AF	A8	2
test	AX, imm16 EAX, imm32	SF, ZF, OF, CF, PF, AF	A9	3 5
test	reg8, imm8	SF, ZF, OF, CF, PF, AF	F6	3
test	reg16, imm16 reg32, imm32	SF, ZF, OF, CF, PF, AF	F7	4 6

助记符	操作数	受影响的标志位	操作码	字节数
test	mem8, imm8	SF, ZF, OF, CF, PF, AF	F6	3+
test	mem16, imm16 mem32, imm32	SF, ZF, OF, CF, PF, AF	F7	4+ 6+
test	reg8, reg8	SF, ZF, OF, CF, PF, AF	84	2
test	reg16, reg16 reg32, reg32	SF, ZF, OF, CF, PF, AF	85	2
test	mem8, reg8	SF, ZF, OF, CF, PF, AF	84	2+
test	mem16, reg16 mem32, reg32	SF, ZF, OF, CF, PF, AF	85	2+
xchg	AX, CX EAX, ECX	无	91	1
xchg	AX, DX EAX, EDX	无	92	1
xchg	AX, BX EAX, EBX	无	93	1
xchg	AX, SP EAX, ESP	无	94	1
xchg	AX, BP EAX, EBP	无	95	1
xchg	AX, SI EAX, ESI	无	96	1
xchg	AX, DI EAX, EDI	无	97	1
xchg	reg8, reg8	无	86	2
xchg	reg8, mem8	无	86	2+
xchg	reg16, reg16	无	87	2
xchg	reg16, mem16	无	87	2+
xlat	无	无	D7	1
xor	AL, imm8	SF, ZF, OF, CF, PF, AF	34	2
xor	AX, imm16 EAX, imm32	SF, ZF, OF, CF, PF, AF	35	3 5
xor	reg8, imm8	SF, ZF, OF, CF, PF, AF	80	3
xor	reg16, imm16 reg32, imm32	SF, ZF, OF, CF, PF, AF	81	4 6
xor	reg16, imm8 reg32, imm8	SF, ZF, OF, CF, PF, AF	83	3
xor	mem8, imm8	SF, ZF, OF, CF, PF, AF	80	3+
xor	mem16, imm16 mem32, imm32	SF, ZF, OF, CF, PF, AF	81	4+ 6+
xor	mem16, imm8 mem32, imm8	SF, ZF, OF, CF, PF, AF	83	3+

助记符	操作数	受影响的标志位	操作码	字节数
xor	reg8, reg8	SF, ZF, OF, CF, PF, AF	32	2
xor	reg16, reg16 reg32, reg32	SF, ZF, OF, CF, PF, AF	33	2
xor	reg8, mem8	SF, ZF, OF, CF, PF, AF	32	2+
xor	reg16, mem16 reg32, mem32	SF, ZF, OF, CF, PF, AF	33	2+
xor	mem8, reg8	SF, ZF, OF, CF, PF, AF	30	2+
xor	mem16, reg16 mem32, reg32	SF, ZF, OF, CF, PF, AF	31	2+

附录 E　80x86 指令（按操作码排列）

助记符	操作数	受影响的标志位[⊖]	操作码	字节数
00	add	mem8, reg8	SF, ZF, OF, CF, PF, AF	2+
01	add	mem16, reg16 mem32, reg32	SF, ZF, OF, CF, PF, AF	2+
02	add	reg8, reg8	SF, ZF, OF, CF, PF, AF	2
02	add	reg8, mem8	SF, ZF, OF, CF, PF, AF	2+
03	add	reg16, reg16 reg32, reg32	SF, ZF, OF, CF, PF, AF	2
03	add	reg16, mem16 reg32, mem32	SF, ZF, OF, CF, PF, AF	2+
04	add	AL, imm8	SF, ZF, OF, CF, PF, AF	2
05	add	AX, imm16 EAX, imm32	SF, ZF, OF, CF, PF, AF	3 5
06	push	ES	无	1
07	pop	ES	无	1
08	or	mem8, reg8	SF, ZF, OF, CF, PF, AF	2+
09	or	mem16, reg16 mem32, reg32	SF, ZF, OF, CF, PF, AF	2+
0A	or	reg8, reg8	SF, ZF, OF, CF, PF, AF	2
0A	or	reg8, mem8	SF, ZF, OF, CF, PF, AF	2+
0B	or	reg16, reg16 reg32, reg32	SF, ZF, OF, CF, PF, AF	2
0B	or	reg16, mem16 reg32, mem32	SF, ZF, OF, CF, PF, AF	2+
0C	or	AL, imm8	SF, ZF, OF, CF, PF, AF	2
0D	or	AX, imm16 EAX, imm32	SF, ZF, OF, CF, PF, AF	3 5
0E	push	CS	无	1
0F 04	shld	reg16, reg16, imm8 reg32, reg32, imm8	SF, ZF, CF, PF OF, AF ?	4

⊖ 问号？表示标志位值可能改变，不过这些值没有意义。

助记符	操作数	受影响的标志位	操作码	字节数
0F 04	shld	mem16, reg16, imm8 mem32, reg32, imm8	SF, ZF, CF, PF OF, AF ?	4+
0F 05	shld	reg16, reg16, CL reg32, reg32, CL	SF, ZF, CF, PF OF, AF ?	3
0F 05	shld	mem16, reg16, CL mem32, reg32, CL	SF, ZF, CF, PF OF, AF ?	3+
0F 80	jo	rel32	无	7+, 3
0F 81	jno	rel32	无	7+, 3
0F 82	jb jnae	rel32	无	7+, 3
0F 82	jc	rel32	无	7+, 3
0F 83	jae jnb	rel32	无	7+, 3
0F 83	jnc	rel32	无	7+, 3
0F 84	je jz	rel32	无	7+, 3
0F 85	jne jnz	rel32	无	7+, 3
0F 86	jbe jna	rel32	无	7+, 3
0F 87	ja jnbe	rel32	无	7+, 3
0F 88	js	rel32	无	7+, 3
0F 89	jns	rel32	无	7+, 3
0F 8A	jp jpe	rel32	无	7+, 3
0F 8B	jnp jpo	rel32	无	7+, 3
0F 8C	jl jnge	rel32	无	7+, 3
0F 8D	jge jnl	rel32	无	7+, 3
0F 8E	jle jng	rel32	无	7+, 3
0F 8F	jg jnle	rel32	无	7+, 3
0F A0	push	FS	无	2
0F A1	pop	FS	无	2
0F A8	push	GS	无	2
0F A9	pop	GS	无	2

助记符	操作数	受影响的标志位	操作码	字节数
0F AC	shrd	reg16, reg16, imm8 reg32, reg32, imm8	SF, ZF, CF, PF OF, AF ?	4
0F AC	shrd	mem16, reg16, imm8 mem32, reg32, imm8	SF, ZF, CF, PF OF, AF ?	4+
0F AD	shrd	reg16, reg16, CL reg32, reg32, CL	SF, ZF, CF, PF OF, AF ?	3
0F AD	shrd	mem16, reg16, CL mem32, reg32, CL	SF, ZF, CF, PF OF, AF ?	3+
0F AF	imul	reg16, reg16 reg32, reg32	OF, CF SF, ZF, PF, AF ?	3
0F AF	imul	reg16, mem16 reg32, mem32	OF, CF SF, ZF, PF, AF ?	3+
0F B6	movzx	reg16, reg8 reg32, reg8	无	3
0F B6	movzx	reg16, mem8 reg32, mem8	无	3+
0F B7	movzx	reg32, reg16	无	3
0F B7	movzx	reg32, mem16	无	3+
0F BE	movsx	reg16, reg8 reg32, reg8	无	3
0F BE	movsx	reg16, mem8 reg32, mem8	无	3+
0F BF	movsx	reg32, reg16	无	3
0F BF	movsx	reg32, mem16	无	3+
10	adc	mem8, reg8	SF, ZF, OF, CF, PF, AF	2+
11	adc	mem16, reg16 mem32, reg32	SF, ZF, OF, CF, PF, AF	2+
12	adc	reg8, reg8	SF, ZF, OF, CF, PF, AF	2
12	adc	reg8, mem8	SF, ZF, OF, CF, PF, AF	2+
13	adc	reg16, reg16 reg32, reg32	SF, ZF, OF, CF, PF, AF	2
13	adc	reg16, mem16 reg32, mem32	SF, ZF, OF, CF, PF, AF	2+
14	adc	AL, imm8	SF, ZF, OF, CF, PF, AF	2
15	adc	AX, imm16 EAX, imm32	SF, ZF, OF, CF, PF, AF	3 5
16	push	SS	无	1
17	pop	SS	无	1
18	sbb	mem8, reg8	SF, ZF, OF, CF, PF, AF	2+
19	sbb	mem16, reg16 mem32, reg32	SF, ZF, OF, CF, PF, AF	2+

助记符	操作数	受影响的标志位	操作码	字节数
1A	sbb	reg8, reg8	SF, ZF, OF, CF, PF, AF	2
1A	sbb	reg8, mem8	SF, ZF, OF, CF, PF, AF	2+
1B	sbb	reg16, reg16 reg32, reg32	SF, ZF, OF, CF, PF, AF	2
1B	sbb	reg16, mem16 reg32, mem32	SF, ZF, OF, CF, PF, AF	2+
1C	sbb	AL, imm8	SF, ZF, OF, CF, PF, AF	2
1D	sbb	AX, imm16 EAX, imm32	SF, ZF, OF, CF, PF, AF	3 5
1E	push	DS	无	1
1F	pop	DS	无	1
20	and	mem8, reg8	SF, ZF, OF, CF, PF, AF	2+
21	and	mem16, reg16 mem32, reg32	SF, ZF, OF, CF, PF, AF	2+
22	and	reg8, reg8	SF, ZF, OF, CF, PF, AF	2
22	and	reg8, mem8	SF, ZF, OF, CF, PF, AF	2+
23	and	reg16, reg16 reg32, reg32	SF, ZF, OF, CF, PF, AF	2
23	and	reg16, mem16 reg32, mem32	SF, ZF, OF, CF, PF, AF	2+
24	and	AL, imm8	SF, ZF, OF, CF, PF, AF	2
25	and	AX, imm16 EAX, imm32	SF, ZF, OF, CF, PF, AF	3 5
27	daa	无	SF, ZF, PF, AF OF ?	1
28	sub	mem8, reg8	SF, ZF, OF, CF, PF, AF	2+
29	sub	mem16, reg16 mem32, reg32	SF, ZF, OF, CF, PF, AF	2+
2A	sub	reg8, reg8	SF, ZF, OF, CF, PF, AF	2
2A	sub	reg8, mem8	SF, ZF, OF, CF, PF, AF	2+
2B	sub	reg16, reg16 reg32, reg32	SF, ZF, OF, CF, PF, AF	2
2B	sub	reg16, mem16 reg32, mem32	SF, ZF, OF, CF, PF, AF	2+
2C	sub	AL, imm8	SF, ZF, OF, CF, PF, AF	2
2D	sub	AX, imm16 EAX, imm32	SF, ZF, OF, CF, PF, AF	3 5
2F	das	无	SF, ZF, PF, AF OF ?	1
30	xor	mem8, reg8	SF, ZF, OF, CF, PF, AF	2+

助记符	操作数	受影响的标志位	操作码	字节数
31	xor	mem16, reg16 mem32, reg32	SF, ZF, OF, CF, PF, AF	2+
32	xor	reg8, reg8	SF, ZF, OF, CF, PF, AF	2
32	xor	reg8, mem8	SF, ZF, OF, CF, PF, AF	2+
33	xor	reg16, reg16 reg32, reg32	SF, ZF, OF, CF, PF, AF	2
33	xor	reg16, mem16 reg32, mem32	SF, ZF, OF, CF, PF, AF	2+
34	xor	AL, imm8	SF, ZF, OF, CF, PF, AF	2
35	xor	AX, imm16 EAX, imm32	SF, ZF, OF, CF, PF, AF	3 5
37	aaa	无	AF, CF SF, ZF, OF, PF ?	1
38	cmp	reg8, reg8	SF, ZF, OF, CF, PF, AF	2
38	cmp	mem8, reg8	SF, ZF, OF, CF, PF, AF	2+
39	cmp	mem16, reg16 mem32, reg32	SF, ZF, OF, CF, PF, AF	2+
3A	cmp	reg8, mem8	SF, ZF, OF, CF, PF, AF	2+
3B	cmp	reg16, reg16 reg32, reg32	SF, ZF, OF, CF, PF, AF	2
3B	cmp	reg16, mem16 reg32, mem32	SF, ZF, OF, CF, PF, AF	2+
3C	cmp	AL, imm8	SF, ZF, OF, CF, PF, AF	2
3D	cmp	AX, imm16 EAX, imm32	SF, ZF, OF, CF, PF, AF	3 5
3F	aas	无	AF, CF SF, ZF, OF, PF ?	1
40	inc	AX EAX	SF, ZF, OF, PF, AF	1
41	inc	CX ECX	SF, ZF, OF, PF, AF	1
42	inc	DX EDX	SF, ZF, OF, PF, AF	1
43	inc	BX EBX	SF, ZF, OF, PF, AF	1
44	inc	SP ESP	SF, ZF, OF, PF, AF	1
45	inc	BP EBP	SF, ZF, OF, PF, AF	1
47	inc	SI ESI	SF, ZF, OF, PF, AF	1

助记符	操作数	受影响的标志位	操作码	字节数
48	dec	AX EAX	SF, ZF, OF, PF, AF	1
48	inc	DI EDI	SF, ZF, OF, PF, AF	1
49	dec	CX ECX	SF, ZF, OF, PF, AF	1
4A	dec	DX EDX	SF, ZF, OF, PF, AF	1
4B	dec	BX EBX	SF, ZF, OF, PF, AF	1
4C	dec	SP ESP	SF, ZF, OF, PF, AF	1
4D	dec	BP EBP	SF, ZF, OF, PF, AF	1
4E	dec	SI ESI	SF, ZF, OF, PF, AF	1
4F	dec	DI EDI	SF, ZF, OF, PF, AF	1
50	push	AX EAX	无	1
51	push	CX ECX	无	1
52	push	DX EDX	无	1
53	push	BX EBX	无	1
54	push	SP ESP	无	1
55	push	BP EBP	无	1
56	push	SI ESI	无	1
57	push	DI EDI	无	1
58	pop	AX EAX	无	1
59	pop	CX ECX	无	1
5A	pop	DX EDX	无	1
5B	pop	BX EBX	无	1

助记符	操作数	受影响的标志位	操作码	字节数
5C	pop	SP ESP	无	1
5D	pop	BP EBP	无	1
5E	pop	SI ESI	无	1
5F	pop	DI EDI	无	1
60	pusha pushad	无	无	1
61	popa popad	无	无	1
68	push	imm16 imm32	无	3 5
69	imul	reg16, reg16, imm16 reg32, reg32, imm32	OF, CF SF, ZF, PF, AF ?	4 6
69	imul	reg16, mem16, imm16 reg32, mem32, imm32	OF, CF SF, ZF, PF, AF ?	4+ 6+
6A	push	imm8	无	2
6B	imul	reg16, imm8 reg32, imm8	OF, CF SF, ZF, PF, AF ?	3
6B	imul	reg16, reg16, imm8 reg32, reg32, imm8	OF, CF SF, ZF, PF, AF ?	3
6B	imul	reg16, mem16, imm8 reg32, mem32, imm8	OF, CF SF, ZF, PF, AF ?	3+
70	jo	rel8	无	7+, 3
71	jno	rel8	无	7+, 3
72	jb jnae	rel8	无	7+, 3
72	jc	rel8	无	7+, 3
73	jae jnb	rel8	无	7+, 3
73	jnc	rel8	无	7+, 3
74	je jz	rel8	无	7+, 3
75	jne jnz	rel8	无	7+, 3
76	jbe jna	rel8	无	7+, 3
77	ja jnbe	rel8	无	7+, 3
78	js	rel8	无	7+, 3

助记符	操作数	受影响的标志位	操作码	字节数
79	jns	rel8	无	7+, 3
7A	jp jpe	rel8	无	7+, 3
7B	jnp jpo	rel8	无	7+, 3
7C	jl jnge	rel8	无	7+, 3
7D	jge jnl	rel8	无	7+, 3
7E	jle jng	rel8	无	7+, 3
7F	jg jnle	rel8	无	7+, 3
80	adc	reg8, imm8	SF, ZF, OF, CF, PF, AF	3
80	adc	mem8, imm8	SF, ZF, OF, CF, PF, AF	3+
80	add	reg8, imm8	SF, ZF, OF, CF, PF, AF	3
80	add	mem8, imm8	SF, ZF, OF, CF, PF, AF	3+
80	and	reg8, imm8	SF, ZF, OF, CF, PF, AF	3
80	and	mem8, imm8	SF, ZF, OF, CF, PF, AF	3+
80	cmp	reg8, imm8	SF, ZF, OF, CF, PF, AF	3
80	cmp	mem8, imm8	SF, ZF, OF, CF, PF, AF	3+
80	or	reg8, imm8	SF, ZF, OF, CF, PF, AF	3
80	or	mem8, imm8	SF, ZF, OF, CF, PF, AF	3+
80	sbb	reg8, imm8	SF, ZF, OF, CF, PF, AF	3
80	sbb	mem8, imm8	SF, ZF, OF, CF, PF, AF	3+
80	sub	reg8, imm8	SF, ZF, OF, CF, PF, AF	3
80	sub	mem8, imm8	SF, ZF, OF, CF, PF, AF	3+
80	xor	reg8, imm8	SF, ZF, OF, CF, PF, AF	3
80	xor	mem8, imm8	SF, ZF, OF, CF, PF, AF	3+
81	adc	reg16, imm16 reg32, imm32	SF, ZF, OF, CF, PF, AF	4 6
81	adc	mem16, imm16 mem32, imm32	SF, ZF, OF, CF, PF, AF	4+ 6+
81	add	reg16, imm16 reg32, imm32	SF, ZF, OF, CF, PF, AF	4 6
81	add	mem16, imm16 mem32, imm32	SF, ZF, OF, CF, PF, AF	4+ 6+
81	and	reg16, imm16 reg32, imm32	SF, ZF, OF, CF, PF, AF	4 6
81	and	mem16, imm16	SF, ZF, OF, CF, PF, AF	4+

助记符	操作数	受影响的标志位	操作码	字节数
		mem32, imm32		6+
81	cmp	reg16, imm16	SF, ZF, OF, CF, PF, AF	4
		reg32, imm32		6
81	cmp	mem16, imm16	SF, ZF, OF, CF, PF, AF	4+
		mem32, imm32		6+
81	or	reg16, imm16	SF, ZF, OF, CF, PF, AF	4
		reg32, imm32		6
81	or	mem16, imm16	SF, ZF, OF, CF, PF, AF	4+
		mem32, imm32		6+
81	sbb	reg16, imm16	SF, ZF, OF, CF, PF, AF	4
		reg32, imm32		6
81	sbb	mem16, imm16	SF, ZF, OF, CF, PF, AF	4+
		mem32, imm32		6+
81	sub	reg16, imm16	SF, ZF, OF, CF, PF, AF	4
		reg32, imm32		6
81	sub	mem16, imm16	SF, ZF, OF, CF, PF, AF	4+
		mem32, imm32		6+
81	xor	reg16, imm16	SF, ZF, OF, CF, PF, AF	4
		reg32, imm32		6
81	xor	mem16, imm16	SF, ZF, OF, CF, PF, AF	4+
		mem32, imm32		6+
83	adc	reg16, imm8	SF, ZF, OF, CF, PF, AF	3
		reg32, imm8		
83	adc	mem16, imm8	SF, ZF, OF, CF, PF, AF	3+
		mem32, imm8		
83	add	reg16, imm8	SF, ZF, OF, CF, PF, AF	3
		reg32, imm8		
83	add	mem16, imm8	SF, ZF, OF, CF, PF, AF	3+
		mem32, imm8		
83	and	reg16, imm8	SF, ZF, OF, CF, PF, AF	3
		reg32, imm8		
83	and	mem16, imm8	SF, ZF, OF, CF, PF, AF	3+
		mem32, imm8		
83	cmp	reg16, imm8	SF, ZF, OF, CF, PF, AF	3
		reg32, imm8		
83	cmp	mem16, imm8	SF, ZF, OF, CF, PF, AF	3+
		mem32, imm8		
83	or	reg16, imm8	SF, ZF, OF, CF, PF, AF	3
		reg32, imm8		
83	or	mem16, imm8	SF, ZF, OF, CF, PF, AF	3+
		mem32, imm8		
83	sbb	reg16, imm8	SF, ZF, OF, CF, PF, AF	3
		reg32, imm8		

助记符	操作数	受影响的标志位	操作码	字节数
83	sbb	mem16, imm8 mem32, imm8	SF, ZF, OF, CF, PF, AF	3+
83	sub	reg16, imm8 reg32, imm8	SF, ZF, OF, CF, PF, AF	3
83	sub	mem16, imm8 mem32, imm8	SF, ZF, OF, CF, PF, AF	3+
83	xor	reg16, imm8 reg32, imm8	SF, ZF, OF, CF, PF, AF	3
83	xor	mem16, imm8 mem32, imm8	SF, ZF, OF, CF, PF, AF	3+
84	test	reg8, reg8	SF, ZF, OF, CF, PF, AF	2
84	test	mem8, reg8	SF, ZF, OF, CF, PF, AF	2+
85	test	reg16, reg16 reg32, reg32	SF, ZF, OF, CF, PF, AF	2
85	test	mem16, reg16 mem32, reg32	SF, ZF, OF, CF, PF, AF	2+
86	xchg	reg8, reg8	无	2
86	xchg	reg8, mem8	无	2+
87	xchg	reg16, reg16	无	2
87	xchg	reg16, mem16	无	2+
88	mov	mem8, reg8	无	2+
89	mov	mem16, reg16 mem32, reg32	无	2+
8A	mov	reg8, reg8	无	2
8A	mov	reg8, mem8	无	2+
8B	mov	reg16, reg16 reg32, reg32	无	2
8B	mov	reg16, mem16 reg32, mem32	无	2+
8C	mov	reg16, sreg	无	2
8C	mov	mem16, sreg	无	2+
8D	lea	reg32, mem32	无	2+
8E	mov	sreg, reg16	无	2
8E	mov	sreg, mem16	无	2+
8F	pop	mem16 mem32	无	2+
91	xchg	AX, CX EAX, ECX	无	1
92	xchg	AX, DX EAX, EDX	无	1

助记符	操作数	受影响的标志位	操作码	字节数
93	xchg	AX, BX EAX, EBX	无	1
94	xchg	AX, SP EAX, ESP	无	1
95	xchg	AX, BP EAX, EBP	无	1
96	xchg	AX, SI EAX, ESI	无	1
97	xchg	AX, DI EAX, EDI	无	1
98	cbw	无	无	1
98	cwde	无	无	1
99	cdq	无	无	1
99	cwd	无	无	1
9A	call	far direct	无	7
9C	pushf pushfd	无	无	1
9D	popf popfd	无	无	1
A0	mov	AL, direct	无	5
A1	mov	AX, direct EAX, direct	无	5
A2	mov	direct , AL	无	5
A3	mov	direct, AX direct, EAX	无	5
A4	movsb	无	无	1
A5	movsw movsd	无	无	1
A6	cmpsb	无	无	1
A7	cmpsw cmpsd	无	无	1
A8	test	AL, imm8	SF, ZF, OF, CF, PF, AF	2
A9	test	AX, imm16 EAX, imm32	SF, ZF, OF, CF, PF, AF	3 5
AA	stosb	无	无	1
AB	stosw stosd	无	无	1
AC	lodsb	无	无	1
AD	lodsw lodsd	无	无	1

助记符	操作数	受影响的标志位	操作码	字节数
AE	scasb	无	无	1
AE	scasw scasd	无	无	1
B0	mov	AL, imm8	无	2
B1	mov	CL, imm8	无	2
B2	mov	DL, imm8	无	2
B3	mov	BL, imm8	无	2
B4	mov	AH, imm8	无	2
B5	mov	CH, imm8	无	2
B6	mov	DH, imm8	无	2
B7	mov	BH, imm8	无	2
B8	mov	AX, imm16 EAX, imm32	无	3 5
B9	mov	CX, imm16 ECX, imm32	无	3 5
BA	mov	DX, imm16 EDX, imm32	无	3 5
BB	mov	BX, imm16 EBX, imm32	无	3 5
BC	mov	SP, imm16 ESP, imm32	无	3 5
BD	mov	BP, imm16 EPB, imm32	无	3 5
BE	mov	SI, imm16 ESI, imm32	无	3 5
BF	mov	DI, imm16 EDI, imm32	无	3 5
C0	rol ror	reg8, imm8	SF, ZF, OF, CF, PF AF ?	3
C0	rol ror	mem8, imm8	SF, ZF, OF, CF, PF AF ?	3+
C0	shl/sal shr sar	reg8, imm8	SF, ZF, OF, CF, PF AF ?	3
C0	shl/sal shr sar	mem8, imm8	SF, ZF, OF, CF, PF AF ?	3+
C1	rol ror	reg16, imm8 reg32, imm8	SF, ZF, OF, CF, PF AF ?	3
C1	rol ror	mem16, imm8 mem32, imm8	SF, ZF, OF, CF, PF AF ?	3+

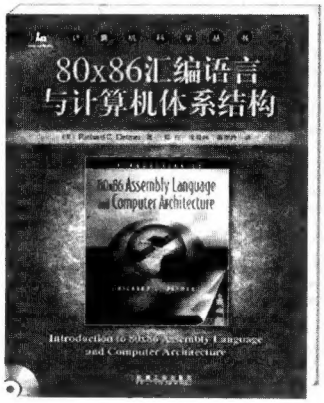
助记符	操作数	受影响的标志位	操作码	字节数
C1	shl/sal shr sar	reg16, imm8 reg32, imm8	SF, ZF, OF, CF, PF AF ?	3
C1	shl/sal shr sar	mem16, imm8 mem32, imm8	SF, ZF, OF, CF, PF AF ?	3+
C2	ret (near)	imm16	无	3
C3	ret (near)	无	无	1
C6	mov	mem8, imm8	无	3+
C7	mov	mem16, imm16 mem32, imm32	无	4+ 6+
CA	ret (far)	imm16	无	3
CB	ret (far)	无	无	1
D0	rol ror	reg8	SF, ZF, OF, CF, PF AF ?	2
D0	rol ror	mem8	SF, ZF, OF, CF, PF AF ?	2+
D0	shl/sal shr sar	reg8	SF, ZF, OF, CF, PF AF ?	2
D0	shl/sal shr sar	mem8	SF, ZF, OF, CF, PF AF ?	2+
D1	rol ror	reg16 reg32	SF, ZF, OF, CF, PF AF ?	2
D1	rol ror	reg16 reg32	SF, ZF, OF, CF, PF AF ?	2+
D1	shl/sal shr sar	reg16 reg32	SF, ZF, OF, CF, PF AF ?	2
D1	shl/sal shr sar	reg16 reg32	SF, ZF, OF, CF, PF AF ?	2+
D2	rol ror	reg8, CL	SF, ZF, OF, CF, PF AF ?	2
D2	rol ror	mem8, CL	SF, ZF, OF, CF, PF AF ?	2+
D2	shl/sal shr sar	reg8, CL	SF, ZF, OF, CF, PF AF ?	2
D2	shl/sal shr sar	mem8, CL	SF, ZF, OF, CF, PF AF ?	2+

助记符	操作数	受影响的标志位	操作码	字节数
D3	rol ror	reg16, CL reg32, CL	SF, ZF, OF, CF, PF AF ?	2
D3	rol ror	mem16, CL mem32, CL	SF, ZF, OF, CF, PF AF ?	2+
D3	shl/sal shr sar	reg16, CL reg32, CL	SF, ZF, OF, CF, PF AF ?	2
D3	shl/sal shr sar	mem16, CL mem32, CL	SF, ZF, OF, CF, PF AF ?	2+
D4 0A	aam	无	SF, ZF, PF OF, AF, CF ?	2
D5 0A	aad	无	SF, ZF, PF OF, AF, CF ?	2
D7	xlat	无	无	1
E0	loopne loopnz	无	无	11+
E1	loope loopz	无	无	11+
E2	loop	无	无	11+
E3	jecxz	rel8	无	2
E8	call	rel32	无	5
E9	jmp	rel32	无	5
EB	jmp	rel8	无	2
F2	repnz repne	无 (string instruction prefix)	无	1
F2 A6	repne cmpsb	无	无	2
F2 A7	repne cmpsw repne cmpsd	无	无	2
F2 AE	repne scasb	无	无	2
F2 AF	repne scasw repne scasd	无	无	2
F3	rep repz repe	无 (string instruction prefix)	无	1
F3 A4	rep movsb	无	无	2

助记符	操作数	受影响的标志位	操作码	字节数
F3 A5	rep movsw rep movsd	无	无	2
F3 A6	rep stosb	无	无	2
F3 A6	repe cmpsb	无	无	2
F3 A7	rep stosw rep stosd	无	无	2
F3 A7	repe cmpsw repe cmpsd	无	无	2
F3 AE	repe scasb	无	无	2
F3 AF	repe scasw repe scasd	无	无	2
F5	cmc	无	CF	1
F6	div	reg8	SF, ZF, OF, PF, AF ?	2
F6	div	mem8	SF, ZF, OF, PF, AF ?	2+
F6	idiv	reg8	SF, ZF, OF, PF, AF ?	2
F6	idiv	mem8	SF, ZF, OF, PF, AF ?	2+
F6	imul	reg8	OF, CF SF, ZF, PF, AF ?	2
F6	imul	mem8	OF, CF SF, ZF, PF, AF ?	2+
F6	mul	reg8	OF, CF SF, ZF, PF, AF ?	2
F6	mul	mem8	OF, CF SF, ZF, PF, AF ?	2+
F6	neg	reg8	SF, ZF, OF, CF, PF, AF	2
F6	neg	mem8	SF, ZF, OF, CF, PF, AF	2+
F6	not	reg8	无	2
F6	not	mem8	无	2+
F6	test	reg8, imm8	SF, ZF, OF, CF, PF, AF	3
F6	test	mem8, imm8	SF, ZF, OF, CF, PF, AF	3+
F7	div	reg16 reg32	SF, ZF, OF, PF, AF ?	2
F7	div	mem16 mem32	SF, ZF, OF, PF, AF ?	2+

助记符	操作数	受影响的标志位	操作码	字节数
F7	idiv	reg16 reg32	SF, ZF, OF, PF, AF ?	2
F7	idiv	mem16 mem32	SF, ZF, OF, PF, AF ?	2+
F7	imul	reg16 reg32	OF, CF SF, ZF, PF, AF ?	2
F7	imul	mem16 mem32	OF, CF SF, ZF, PF, AF ?	2+
F7	imul	mem16 mem32	OF, CF SF, ZF, PF, AF ?	4 6
F7	mul	reg16 reg32	OF, CF SF, ZF, PF, AF ?	2
F7	mul	mem16 mem32	OF, CF SF, ZF, PF, AF ?	2+
F7	neg	reg16 reg32	SF, ZF, OF, CF, PF, AF	2
F7	neg	mem16 mem32	SF, ZF, OF, CF, PF, AF	2+
F7	not	reg16 reg32	无	2
F7	not	mem16 mem32	无	2+
F7	test	reg16, imm16 reg32, imm32	SF, ZF, OF, CF, PF, AF	4 6
F7	test	mem16, imm16 mem32, imm32	SF, ZF, OF, CF, PF, AF	4+ 6+
F8	clc	无	CF	1
F9	stc	无	CF	1
FC	cld	无	DF	1
FD	std	无	DF	1
FE	dec	reg8		2
FE	dec	mem8	SF, ZF, OF, PF, AF	2+
FE	inc	reg8	SF, ZF, OF, PF, AF	2
FE	inc	mem8	SF, ZF, OF, PF, AF	2+
FF	call	reg32 (near indirect)	无	2
FF	call	mem32 (near indirect)	无	2+
FF	call	far indirect	无	6
FF	dec	mem16 mem32	SF, ZF, OF, PF, AF	2+
FF	inc	mem16 mem32	SF, ZF, OF, PF, AF	2+

助记符	操作数	受影响的标志位	操作码	字节数
FF	jmp	reg32	无	2
FF	jmp	mem32	无	2+
FF	push	mem16 mem32	无	2+



80X86汇编语言与计算机体系结构

中文版： 978-7-111-17617-0
丛书名：计算机科学丛书
作者： Richard C.Detmer
译者： 郑红　庞毅林　蒋翠玲
定价： 49.00元

英文版： 978-7-111-15312-X
定价： 55.00元

Intel微处理器（原书第7版）

中文版： 978-7-111-22827-1
丛书名：计算机科学丛书
作者： Barry B. Brey
译者： 金惠华　艾明晶　尚利宏
定价： 85.00元

计算机组成与体系结构

中文版： 978-7-111-19048-3
丛书名：计算机科学丛书
作者： Linda Null, Julia Lobur
译者： 黄河　等
定价： 55.00元

英文版： 978-7-111-15311-1
定价： 55.00元